Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Institut für Praktische Informatik und Medieninformatik
Fachgebiet Datenbanken und Informationssysteme

Diplomarbeit

CouPé: Ein Query Processor für UniStore

CouPé: A Query Processor for UniStore

Diplomarbeit zur Erlangung des akademischen Grades Diplominformatiker,
vorgelegt der
Fakultät für Informatik und Automatisierung der
Technischen Universität Ilmenau
von:

Martin Richtarsky
Matr.-Nr.: 26356

Verantwortlicher Professor:	Prof. Dr.-Ing. habil. Kai-Uwe Sattler
Hochschulbetreuer:	Dipl.-Inf. Marcel Karnstedt
Datum:	Ilmenau, 13. März 2007
Inventarisierungsnummer:	2006-12-13/145/IN97/2254

Abstract

Due to the “information revolution”, an ever-increasing volume of data needs to be processed. Traditional databases often are not up to these tasks, as they do not scale well, are expensive and require administration. Peer-to-peer (P2P) databases promise database-like querying functionality on top of massively scalable P2P systems. Participants of such systems contribute resources and queries are processed without any central coordination by cooperation of the peers. This thesis implements a query processor using these concepts on top of P-Grid, a DHT (Distributed Hash Table) P2P network. Building on a simple implementation, various features are added to improve performance. Different types of operators are created which make use of these features and the available indexes to provide alternative ways of evaluating logical operators. In addition to scalability, an emphasis is placed on fault tolerance, similarity and schema operations. The system is evaluated on PlanetLab, a large-scale research network. The feasibility of the concept is demonstrated and detailed results w.r.t. the available operators are presented. Possible future operators and other improvements of the system are discussed.

Contents

1 Introduction

1.1 Motivation

The “information revolution” has led to a dramatic increase in the volume of data that needs to be processed by databases. Search engines like Google own large data centers with vast amounts of processing and storage capacity to handle the challenges of search. Data mining has become essential to business. In this context, two key problems arise: traditional databases, even distributed databases, often rely on central components which prohibit scaling beyond a certain point and can often not be used in such scenarios. Also, initial investments and operating costs for large-scale processing facilities are huge – the achievable computing power is strictly limited by the available budget.

The peer-to-peer (P2P) approach shows a way out of this dilemma. In the 1990s, Internet users banded together as equals – peers – in the Napster network to share music. Due to the large number of participants, many songs were available for download. The system itself was not strictly P2P, which led to its eventual demise for legal reasons. Nowadays, BitTorrent [Bit07] and IP-TV applications like SopCast [Sop07] follow a more strict P2P approach to achieve the same purpose: using “power in numbers” to do things a single instance is hardly capable of, in this case the provision of vast bandwidth for data distribution. Existing infrastructure can be used and central components are only required for bootstrapping. While these systems are often used for illegal purposes, this should not taint their potential for legitimate applications.

Applying these success stories to the database world would solve the problems outlined above and provide a generic, enabling technology for public data management. “Public” means that the data is provided and accessed by many parties, for example, multiple research institutions or individuals. Distributed Hash Tables (DHTs) have been heavily researched in the past years. They use the P2P paradigm, do not have central components and provide scalable distributed storage and localization of data with simple primitives. Extending them to offer rich database-like query semantics is a promising approach to arrive at a massively scalable data management system. Concrete ap-

plications are specialized search engines, directory services and all scenarios where many users need to publish and access data and a central solution is not feasible either for cost or scalability reasons. For example, participating research institutions could install such a system in computer labs and contribute idle resources to form a powerful distributed system for indexing and querying of research data.

UniStore [KSR+07] is a project at Technische Universität Ilmenau to create such a data management system. An important part of it is the query processor, which accepts requests posed in a query language and processes the results. The concepts, design, implementation and evaluation of key parts of it are the focus of this thesis.

1.2 Objective

The main objectives of this thesis are:

- The creation of a query processor for a dynamic P2P environment, integrating existing components.
- Deployment and evaluation of the processor on the Internet to determine the feasibility, scalability and properties of the system.

The processor will be built on top of P-Grid, a DHT, described in detail in sec. 2.2. Challenges like dynamically joining and leaving hosts and also the failure of peers are already addressed by this system. The query processor must be highly scalable and efficient. Starting from a query posed in the Vertical Query Language (VQL) for which a parser already exists, the required data must be located in the network, processed with the correct operators and the results delivered to the initiating peer. Missing data must be tolerated, as no central instance can control data quality. The processing load should be balanced if possible. Therefore, good processing strategies must be found. There are also many ways to implement a particular operator (for example, a join), so multiple versions should be available for choice by an optimizer[1]. In search engines or directory services fuzzy search is an important feature. Therefore, similarity queries should be supported. As a public data management system stores data from many sources, heterogenous schemata exist. To efficiently deal with them, operators should work on schema level.

[1] optimization itself is out of the scope of this work

Strict ACID guarantees as known from traditional databases are not required. Implementing them would require complicated protocols or central components which are not suited to the P2P approach and not required by most of the applications the system targets.

The evaluation must demonstrate the feasibility of the concept of a P2P query processor. The performance of the processing strategies and operator implementations has to be examined and situations when they perform best identified.

1.3 Outline of Thesis

Chapter 2 introduces P2P networks and shows two basic approaches to data localization. P-Grid, the network used for UniStore, is examined in detail. State of the Art-approaches to bring query processing to DHT networks are reviewed in chapter 3. Key parts of UniStore on which this work is based are discussed in chapter 4. This includes techniques for storing structured data in P-Grid and the user interface to the query processor, the Vertical Query Language. The next chapter takes a detailed look at the challenges faced by a query processor in a P2P system, presents solution and a design for it. Chapter 6 discusses the key component of this design, the execution engine, including execution strategies and ways to determine when a query has been completed. The operators implemented as part of the processor are described in the following chapter and possible future operators are discussed. The implementation is documented in chapter 8. Another key part of this thesis is the evaluation of the system on PlanetLab (chapter 9). Conclusions as well as ideas for future research can be found in the final chapter.

2 Introduction to P2P Networks and P-Grid

The P2P paradigm is key to this work. This section gives an overview of the most important concepts and properties. Two contrasting data localization techniques in P2P are presented and selected designs are categorized. Lastly, the P-Grid network, on top of which the query processor will be implemented, is discussed in detail.

2.1 P2P Networks

Client-server architectures provide only limited scalability, flexibility and have single points of failure (the server). Another approach which is often a better fit is the peer-to-peer (P2P) paradigm. As the name implies, all the participating peers are “equal”. In the purest form, no central components exist. Key properties are [SHS05]:

- Each peer can act as a server and a client, providing access to resources or accessing resources on other peers.
- No central coordination or central storage exists.
- Each peer only has limited knowledge about the system, no global knowledge is available.
- The global behaviour of the system emerges as the sum of all local interactions between peers.
- A fully connected topology is not required, it is also possible to establish links between peers by routing through other peers.

Because they establish a logical link structure between peers which need not depend on the underlying network and route messages based on these logical connections, but by means of the base network, P2P systems are often called *overlay networks*.

While the P2P approach has been known and employed for quite some time[1], it took file-sharing applications like Napster or Gnutella to make it popular. A large user-base showcased the potential of this architecture and sparked interest in academia. Weaknesses in early designs were identified and fixed, leading to many interesting and powerful P2P applications. All kinds of resources can be harnessed:

Processing power Processing load can be distributed among peers (computing grids).

Storage Large-scale distributed storage can be used for demanding scientific applications or public data management, like the semantic web.

Bandwidth Unused bandwidth can be utilized to transport data to other peers in the network. Examples are BitTorrent [Bit07] and IP-TV applications like SopCast [Sop07].

Information sources Sensors or other information sources can form P2P networks.

P2P can be a fair way to share costs between all users. Consider a scenario where a not-for-profit organization provides a documentation for download. Instead of having to pay for the downstream bandwidth themselves, they can make use of the downloaders' unused upstream bandwidth by offering the file via a BitTorrent tracker. Distributing CPU-intensive calculations among many peers is motivated by the fact that many separate computers with moderate CPU power are cheaper than one "super computer". In the context of this work, the P2P concept is attractive for storage and CPU load distribution.

Existing P2P designs can be classified in many ways. One important aspect is how data items in the network are located: query messages must be routed to those peers responsible for the requested items in an efficient way. P2P systems can be classified by the type and amount of "structure" in the form of routing indexes they employ for this task. One extreme are broadcast-based approaches without any indexes, the other are DHTs which guarantee that all matching data available in the network is returned. They also provide theoretical upper bounds on the number of hops required. These two extremes will be illustrated in the next two sections. Most P2P systems fall somewhere inbetween them.

[1] examples are the ARPANET and the feed exchanges between USENET servers

2.1.1 Broadcast Networks

Query messages to locate data objects are flooded through the network. At each peer they are processed and local results returned to the initiator. To prevent overloading of the network, a maximum hop count limits the message to a "horizon". Therefore, not all results might be returned – which is a problem for rare items that are available at a few peers only. The advantage is that the query semantics are not limited in any way because routing is query-independent in these broadcast-systems. This allows for arbitrarily complex queries.

2.1.2 DHT Networks

Routing of messages is based on some kind of index: DHTs store data items addressed by a key like a hash table, but distributed among many peers. The key space of the hash function is partitioned and each peer is responsible for one slice. If the current peer is not responsible for the request, a greedy routing strategy similar to IP routing is used to forward the message towards the destination with a minimal number of hops. For most DHT designs it is possible to derive theoretical bounds for the number of hops (usually $O(logN)$ for a network with N peers or key space partitions) and the size of the routing tables.

2.1.3 Examples

The original Gnutella design is a broadcast P2P network. Scalability issues caused by flooding traffic were addressed by only flooding to a certain number of neighboring peers, a unique ID to eliminate already-seen query messages and the time-to-live flooding "horizon". The FastTrack technology, used by KaZaA and Grokster, and later Gnutella designs improved on these ideas by introducing "super peers": it was observed that certain peers were bottlenecks (for example, because of slow dial-up connections), while others had much more resources to contribute to the network. These powerful nodes become super peers and communicate with other super peers just like in the original Gnutella design – they are responsible for query processing. "Normal" peers register their resources with the super peers and use them as proxy for query processing, instead of talking to other peers themselves. Schema-based P2P databases like Edutella [NWQ+02] enhance their local knowledge with routing indexes that indicate what data is reachable through which neighboring peers. Influential DHT designs are

Chord [SMK+01], CAN [RFH+01] and P-Grid [Abe01]. These proposals have much in common: they share the same hashtable API (`put()` and `get()` operations), theoretical bounds for the number of hops and the size of the routing tables, and protocols for joining and leaving the network. Main differences are the topology used, the way the routing tables are set up (i.e., which peers are known to the current peer), and the specific theoretical bounds [SHS05].

2.2 The P-Grid Network

The P2P network used in this work is P-Grid [Abe01], a DHT. Binary keys reference data items in the hash table. The keys are created from application keys[2] by user-defined hash functions. At the heart of P-Grid is a virtual binary search tree with the peers located at the leaves, depicted in fig. 2.1. A peer's *path* is the binary number created by walking from the root of the tree to the peer, appending 0 for each left subtree and 1 for each right subtree visited. The peer is responsible for all binary keys starting with this prefix and manages the associated values in its data store. Multiple peers can exist for one path to provide replication. Each peer has a routing table for forwarding messages it is not responsible for: for each prefix p of the peer's path ("level") it contains references to peers that have the same p, but with the last bit inverted. In fig. 2.1 the table is shown with one entry for each level at the bottom of the peers. For example, A is responsible for all keys starting with 00. To handle all possible keys, it only needs to know where to send queries for keys starting with 1 and 01. At routing time, a greedy algorithm is used to find the level with the longest common prefix in relation to the destination path and a random host is selected from it. The query in fig. 2.1 further illustrates this: an application on peer F starts a request for key "as" which is hashed to the P-Grid key 100 and sent to the DHT on the peer for processing. Peer F is not responsible for the prefix, so it consults its routing table and determines the level with the longest common prefix (1). It forwards the message to E (instead of E, C or D could have been listed here as well) which determines D as the next best match. D stores the requested key, since its path is a prefix of 100, and answers the request to F which forwards it to the application.

The virtual search tree is created and maintained automatically by P-Grid so that each peer is responsible for roughly the same number of keys. This happens with a random walk strategy: peers contact eachother randomly and exchange information about the data stored. Data items can be transfered to the partner for better load balancing and

[2] for example, an ID

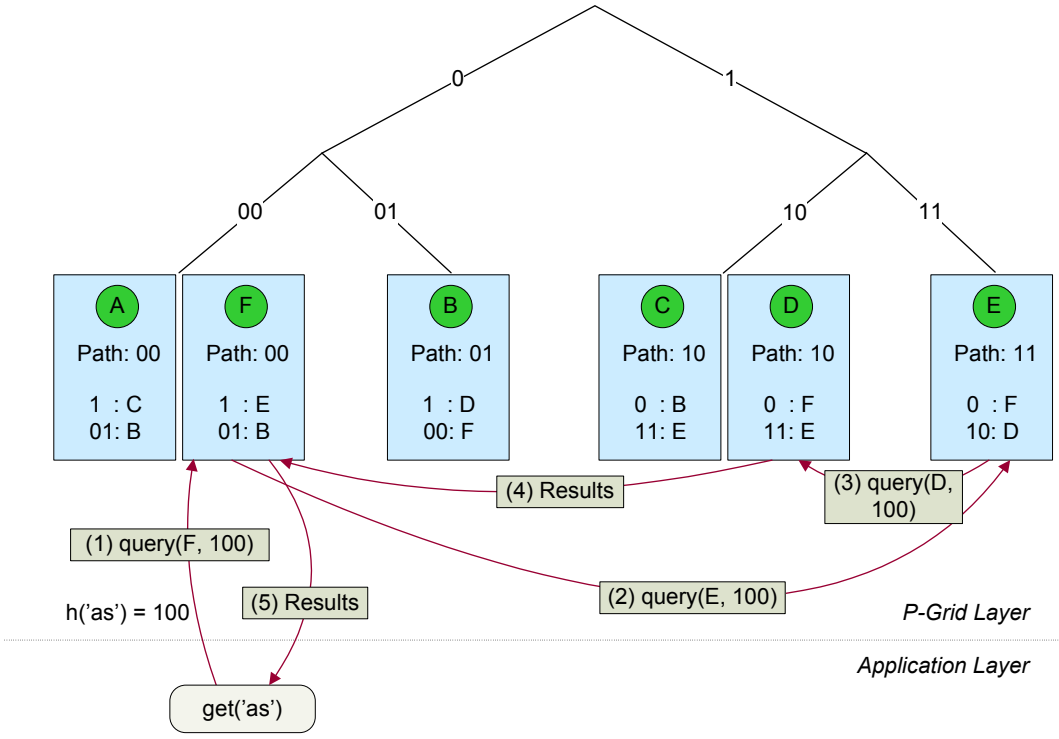

Figure 2.1: P-Grid virtual tree and routing of a message (adapted from [DHJ+05])

the paths and routing tables are updated accordingly. Dedicated bootstrapping peers provide addresses of established peers to peers just joining the grid [ADHS05b].

This process ensures storage load balancing for nearly any data distribution. In particular it makes it possible to use *order-preserving hashing*, which maps similar application keys to similar P-Grid binary keys, in contrast to randomized hashing. Thus, similar keys are mapped to the same or a neighboring[3] peer and ranges of keys can be accessed efficiently. This scheme is usually not feasible for normal DHTs as it causes storage load imbalance: for skewed data distributions, relatively few peers can become responsible for a majority of the keys in the network. P-Grid adjusts the paths of each peer to alleviate this problem [ADHS05b]. One issue remains – identical application keys will still be mapped to identical binary keys and therefore stored on the same peer. When extreme data distributions are to be expected, it might be useful to append a random value to each key, enabling P-Grid to handle this case as well. The original data can still be extracted with a *range query* on the key, which will be discussed next.

Range queries extract all data stored in the range between two binary P-Grid keys, like 101000-101010[4]. P-Grid routes the request to all peers responsible using an efficient, parallel *shower algorithm* [DHJ+05]. In the context of this work, range queries are mainly used for accessing all keys with a common prefix, so the term "prefix queries" will also be used. One application of such a query has been mentioned above. Others become possible by usage of a prefix-preserving hash function h for key generation. For two application keys a, b and the *is-prefix-of* relationship $\subseteq$, h has the following property:

$$a \subseteq b \Rightarrow h(a) \subseteq h(b)$$

This way the prefix relationship between two keys remains after hashing to P-Grid's key space. Therefore, a prefix query can be used to query for prefixes of application keys. Prefix string searches and integer range queries can be implemented on top of this, other applications will be discussed in sec. 4.1.1 and sec. 6.4.1.

In addition to the multiple references stored in the routing tables a configurable number of replicas for each peer provide fault tolerance in the face of peer and network failures, as well as load balancing. In fig. 2.1, peers A, F and C, D are responsible for the same path. Epidemic algorithms ensure replica consistency.

[3] which can be defined by the proximity of the peers at the leaf level of the tree
[4] a lexicographic-like ordering is assumed

Available data is guaranteed to be found in $O(logN)$ time for N distinct paths ("key space partitions") in the grid (direct key lookup) [DHJ+05]. Routing tables can require up to $O(N)$ space, but only for heavily skewed data distributions, so this poses no problem in practice [ADHS05a].

With these properties, P-Grid provides a good basis for a P2P storage system and an accompanying distributed query processor. In contrast to broadcast-based systems data will always be found when there are no peer failures. These can be alleviated by replication, which also provides query load balancing for free. The known logarithmic bounds make it possible to build a cost model to be used by a query planner. Furthermore, they guarantee that the system will scale in the number of peers. Finally, P-Grid already offers efficient range queries, while other DHT designs must be enhanced to handle them.

More information about P-Grid, including papers on a wide range of topics, can be found on the project website [Con07]. The foundations for P-Grid were presented in [Abe01]. Good overviews of the current system are given in [ACMD+03] and [ADHS05a] (German). Range queries and their costs are discussed in [DHJ+05]. The Java implementation of P-Grid can be obtained from the authors on request.

3 Related Work: DHT Query Processing/Databases

DHT query processing can be seen as the next logical step from shared-nothing query processing, which was quite successful (for a good overview of these systems, see [DG92]). The big difference is that no central knowledge or components are used: no global schema must be maintained and transaction monitors or central query processors do not exist. While this has some drawbacks – it is harder to process queries optimally, no ACID guarantees can be provided – it allows for much greater scalability. In the systems presented here, a distributed data structure, the DHT, is used to spread storage and processing load across many peers and to efficiently route messages between them. Upper bounds for routing hops or routing table size are known for the underlying DHT designs, thus making it possible to assess performance before deployment.

Auxiliary Research Plenty of research has focused on solving some of the *specific* problems encountered when trying to enhance DHTs with complex querying capabilities. There are many interesting proposals for range queries, aggregation, substring search and other problems. These will not be discussed here as query processing in general is the focus of this chapter and work. A comprehensive overview of such research is available in [RM06].

3.1 PIER

PIER [HCH+05, HHL+03] is an extensive effort to build a large-scale DHT query processing engine. It is DHT-agnostic and has been used with CAN [RFH+01], Chord and Bamboo [RGRK04]. Tuples are annotated by table name, column names and column types. Primary indexes are created by inserting all tuples into the DHT with a partition attribute serving as the key. PIER supports range queries by means of a Prefix

Hash Tree [RHS03] and broadcasting of queries to all nodes with a virtual tree of all peers. Tuples are only stored in the primary index. Secondary indexes can be created by referencing the tuple ID. PIER does not offer persistent storage, it uses a *soft-state* approach: publishers must periodically refresh their data to keep it from expiring. This acts as a garbage collector, but places additional load on the system.

Query plans consist of multiple operator graphs (opgraphs) which represent locally connected dataflow operators. These opgraphs are distributed to the appropriate peers and exchange data using the DHT as storage ("DHT rehashing"). Implemented operators include selection, projection, join, group-by, tee, union, duplicate elimination and Eddies [AH00]. Supported join algorithms are symmetric hash join and fetch matches join. Symmetric semi-join and bloom filter join rewriting strategies are used to reduce bandwidth consumption. One example of PIER's distributed operators is the (equi) hash join which uses the DHT as the hash table: all tuples from both tables are republished in the DHT with the values of the join columns as keys. This way, tuples with identical values arrive at the same peer which computes the subjoin locally and forwards the result. This makes it possible to process large joins. Furthermore, hierarchical algorithms for aggregations and joins are presented which operate in a similar way but use the virtual tree mentioned above. Snapshot and continuous queries are supported. Timeouts define the end of processing.

PIER has been evaluated thoroughly in simulation and also in the "real world". In [LHH+04] the authors describe PIERSearch, an extension for the Gnutella network. As keyword queries for rare items often fail to provide results, they augmented the standard Gnutella flooding-based querying approach with a DHT-based index which they queried for rare items. This hybrid system improved performance and recall. The number of Gnutella queries without result were reduced by 18%.

In contrast to PIER, rehashing will not be used in this work because it introduces delays and is not ideal for interactive applications. Instead, P-Grid's prefix queries are used among other techniques to efficiently parallelize queries and distribute processing load.

3.2 "Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks"

This paper [TP03] presents a Chord-based [SMK+01] query processing framework. Each tuple is indexed by hashing its distinct identifier[1] and, for each attribute, by applying an order-preserving hash function for efficient range queries. To alleviate the overhead of these $k + 1$ replicas, the authors suggest storing references to the tuples for the attribute indexes, similar to PIER. The used hash function might lead to imbalanced storage loads for skewed data distributions. As pointed out in sec. 2.2, P-Grid eliminates this problem.

Rudimentary and non-rudimentary queries are categorized according to the operators, number of attributes and relations involved. These include selection (equality), range queries, joins and aggregation/grouping. Algorithms for solving the rudimentary queries are presented. They provide the basis for processing the non-rudimentary queries. Only integers are supported as data type. For range queries and joins, the authors propose *range guards*. For each attribute frequently involved in range queries or joins, a number of range guard-peers are defined which also store the tuples in an order-preserving way like the attribute indexes, but distributed over fewer peers. Adjacent range guards are linked to each other. For l range guards and N total peers, the worst-case number of routing hops decreases from $O(N)$ to $O(l)$. Another optimization is made for multi-attribute selection queries by hashing tuples for all attributes involved in the selection simultaneously. This makes it possible to find matches with a single key lookup.

In contrast to this work, queries on schema level and similarity queries are not considered and no evaluation is given.

3.3 RDFPeers

RDFPeers [CF04] uses a triple-based data model comparable to the one used in this work (see sec. 4.1) to store RDF data. Indexes exist for each component of an RDF triple (subject, predicate, object). String data is hashed with SHA-1, numeric attributes with a locality-preserving hash function. The underlaying network used is MAAN [CFCS04],

[1] for example, the primary key

an extension of Chord, which can also answer multi-attribute and range queries. A native query resolver utilizes the MAAN to provide results. Query modules can be used to map more formal query languages to these native queries and the authors plan to implement RDQL [MSR02] in this fashion. To show that this is feasible, they explain how to process specific RDQL queries. Supported native queries are:

- "atomic", exact-match queries that specify a value for zero, one, two or all three components of the RDF triple
- disjunctive queries for specifying sets of values to match
- range queries for numeric objects
- conjunctive queries for a common subject variable

Theoretical upper bounds for these queries as well as simulation results are provided. Load-balancing problems are identified and a solution for minimizing storage imbalance is presented. It is possible to replicate data by an adjustable factor to enhance resilience.

No provisions are made for supporting minimum/maximum or similarity queries, which will be covered in this thesis.

3.4 "A Query Processor for CAN-based P2P Systems"

[Rös05] discusses query processing of relational data in CAN [RFH+01]. Two different data distribution strategies are evaluated: the first stores a complete relation on a single peer (only the relation ID is hashed), the second distributes data along a Z curve in the CAN overlay network. A locality-preserving hash function is used for this. In combination with an extension made to CAN, which makes it possible to send messages to all peers responsible for data in a given interval on the Z curve, this enables efficient range queries.

The following operators were implemented:

- Projection
- Selection (equality, range queries on numeric data, edit distance on string data)
- two Join variants: Symmetric Hash Join, Ship Where Needed-Join

- two Grouping/Aggregation variants: a central operator that delegates processing to a single peer and a parallel implementation that rehashes the tuples to the CAN

Similar to the MQP concept [PM02b] (also see sec. 6.1), query plans are routed through the network and data is inserted in the plan during processing to be consumed by other plan operators. Plans are processed in post-order, with the exception of the symmetric hash join implementation, where the child operators are started in parallel. The system is evaluated with a CAN simulator and detailed results are provided for different operator implementations, the two data distribution strategies and under load. However, no "real world" tests were performed. Schema-level operations, minimum/maximum or similarity joins are not supported. This work implements such complex operators and uses PlanetLab [CCR+03] for "real world" tests.

3.5 "Similarity Queries in P-Grid-based P2P Networks"

Query processing in a P-Grid network is researched in [Wie06]. Some of the results presented lay the foundations for this thesis. Different storage schemes were considered and a triple-based model was chosen. All triples are indexed on the attribute in conjunction with the value and on the object ID. Q-gram indexes enable similarity queries on strings. A detailed description of this scheme is provided in sec. 4.1.2. Supported operators include:

- Selection (equality, similarity)
- Join (equality, similarity)
- Minimum/Maximum (for integer data)
- Nearest Neighbor
- Substring Search

Different versions of these operations were implemented and compared on PlanetLab internet nodes. The goal of this thesis was to simulate and test several strategies in order to solve some problems on the way to complex query processing. As such, it does not support query plans nor flexible combinations of operators in general, which is the focus of this thesis.

4 Foundations

This chapter documents foundations for the query processor. The first section deals with the partitioning of structured data for distributed storage and efficient access in a DHT. Indexes for exact-match and similarity access are presented. In the second section, querying of such data is discussed. To execute a query, the fragmented data must be fetched from the indexes, recomposed and processed. An algebra is presented which defines the logical operators to accomplish this. Lastly, a query language using this algebra is introduced.

4.1 Storing Structured Data in a DHT

A key problem on the way to a massively distributed, DHT-based database is the efficient storage of structured data, which will be relational data in the case of UniStore. This is because DHTs only support basic hash table operations:

- `put(key, value)`: associate key with value
- `value = get(key)`: retrieve the value associated with key

Based on these primitives, a partitioning scheme is presented in [KSHS06b, Wie06] which provides the basis for this work. Fig. 4.1 shows the partitioning of a table with three rows and columns. Each element ("value") of each tuple is annotated with the corresponding column name ("attribute") and object identifier ("OID"), which must be unique on a tuple basis, to form nine triples of the form $(OID, attribute, value)$[1] (abbreviated as (OID, att, val)). OID is of type integer, att a string and val can be a string or integer. Efficient access structures ("indexes") are created by hashing certain elements of the triples and storing them at the corresponding peer with `put()`. Two kinds of indexes will be discussed in the next two sections. Note that in contrast to indexes found in traditional databases, every index contains the triple itself, not just a pointer to it.

[1] similar to RDF triples

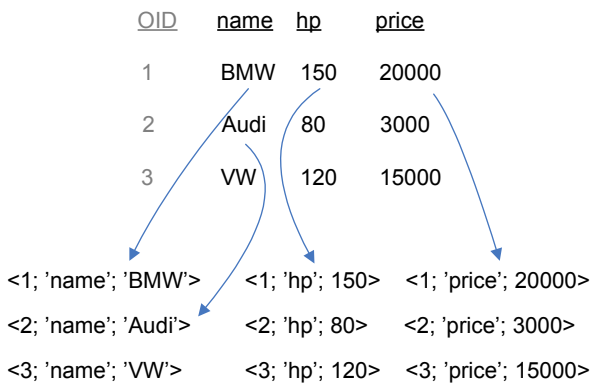

Figure 4.1: Converting structured data to triples

Notation In the following, concatenation is denoted by "$\circ$". Some of the logical operators discussed in sec. 4.2.2 will already be mentioned here (typeset in $italics$). They are equivalent to the logical operators known from database literature and define operations on data, for example, extracting data ($Extraction$) or filtering of data ($Selection$). CouPé must provide implementations for them which make use of the indexes described below. h represents a prefix-preserving hash function responsible for hashing application keys[2] to the binary keys used in P-Grid. This special property can be expressed as ($\subseteq$ denotes the *is-prefix-of* relationship):

$$a \subseteq b \Rightarrow h(a) \subseteq h(b)$$

4.1.1 Exact-match Indexes

These indexes are used to implement the $Extraction$ and $Materialization$ operators.

OID Index For each triple, $h(OID)$ is calculated and the triple is stored at the peer responsible for this key. Therefore, all triples belonging to the same tuple will end up on the same peer, allowing fast reconstruction of tuples with only one member triple or even just the OID available. This allows the implementation of $Materialization$ (sec. 4.2.2).

[2] integers or strings

Attribute-Value Index (AV Index) Each triple is inserted at $h(att) \circ h(val)$. The index can be utilized in four ways, the P-Grid key(s) used for lookup are shown in parentheses:

- extract all stored data (a prefix query with an empty key)
- attribute extraction (a prefix query on $h(att)$)
- exact-match on a given attribute/value ($h(att) \circ h(val)$)
- integer range queries on a given attribute (a range query on $h(att) \circ h(lowerBound) - h(att) \circ h(upperBound)$)

The first query simply fetches all data from the index. The second one limits this operation to one column and returns all triples with this attribute name, the last two additionally place constraints on the values. Range queries on a given attribute are similarity queries: for a given integer value and a maximum distance, the lower and upper bounds can be computed and used as range query parameters. This approach can not be used for string similiarity searches, because the hash function does not represent distance information between two strings. An alternative approach will be presented in sec. 4.1.2. The AV index is used for implementing $Extraction$ (sec. 4.2.2) and $Materialization$ (sec. 4.2.2).

Value Index Indexing each triple on $h(val)$ enables exact-match and integer range queries on values irrespective of the attribute they belong to. This allows efficient retrieval of all data with a given value, no matter in what part of the schema it is stored. In this work the index will not be used.

Fig. 4.2 shows the three indexes for three inserted triples (`<1; 'name'; 'BMW'>`, `<1; 'hp'; 150>`, `<2; 'hp'; 80>`) including the binary P-Grid keys as generated by UniStore's hash functions. Peer A stores all keys starting with 0, B all with 1. The underlined triple components indicate the input to the hash function which generated the binary key. The storage space required by each of these indexes is $O(n)$ for n triples.

4.1.2 Similarity Indexes

Up to now no useful method for similarity string search, a requirement from sec. 1.2, has been provided. Only prefix searching is possible by using a prefix-preserving hash function and a prefix query (sec. 2.2). Otherwise, all candidate data must first be extracted and then filtered, which is inefficient. *Q-grams* are an indexing approach for string data which supports similarity access. A q-gram is a substring of length q which can be used as an access key.

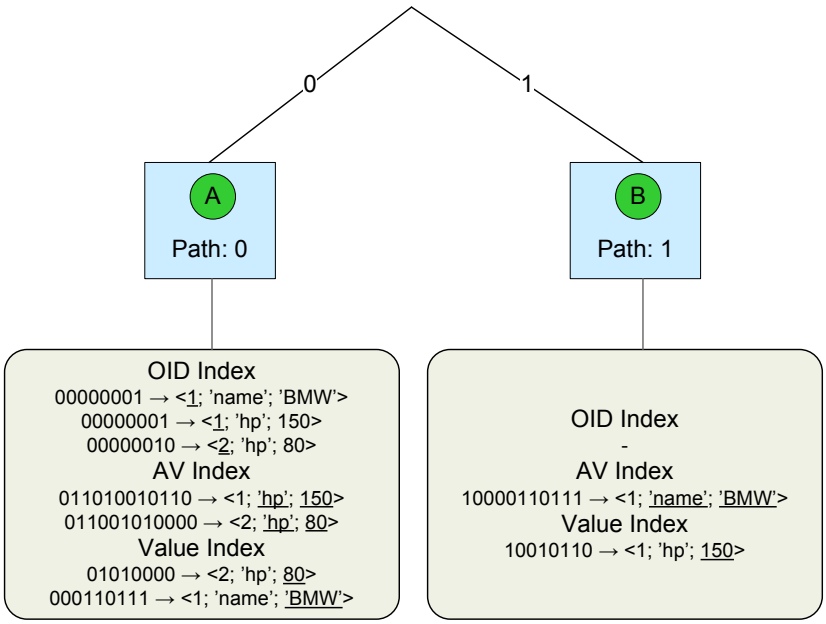

Figure 4.2: Three triples indexed by OID, AV and Value distributed over two peers

Similarity

In the following, the *Levenshtein distance metric* will be used[3] for specifying similarity. Two strings s_1 and s_2 are in distance k when k is the minimum number of operations needed to transform s_1 to s_2. Possible operations are insertion, deletion or substitution of one character. Consider these examples:

$$dist(mistake, miss\underline{t}ake) = 1$$

$$dist(mi\underline{st}ake, mit\underline{s}ake) = 2$$

In the first, one deletion transforms the second string into the first one. In the second example, two operations are needed, either two substitutions, or a deletion with an insertion.

[3] also called edit distance

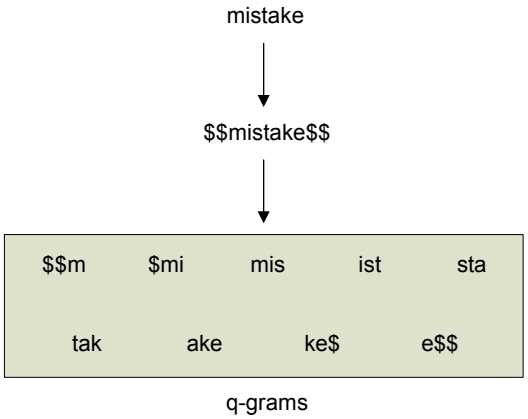

Figure 4.3: Generation of q-grams from a string

Q-gram Generation

Q-grams are generated as follows: first, a value for q must be chosen, $q = 3$ will be used in this work. For each string which should be searchable, such as the attribute or a string value of a triple, $q - 1$ special characters not present in the alphabet (here: $) are appended at the start and the end. The string is converted to lower case because case-sensitivity is usually not desired for similarity search. Then, all possible substrings of length q are generated ($l + 2$ for a string of length l). Fig. 4.3 illustrates this.

Q-gram Indexes

Each q-gram is hashed to $k_n = h(q_n)$ and the corresponding triple stored at this key in the DHT, and similarity search using Lehvensthein distance can be implemented on top of this. Consider a search for a string s with distance $d = 0$ (only identical strings shall be returned). The q-gram generation algorithm as outlined above is applied to the string, including extension with a special character. Any generated q-gram $q_j(s)$ can now be used to find potential matches by querying the DHT for $h(q_j)$. False positives have to be filtered out, as the chosen q-gram can also be part of other strings. Fig. 4.4 shows two triples where the values have been indexed and a query for “misstake” . The chosen q-gram is “$mi”. This locates both triples, but only the second one passes the distance metric. For a query with $d = 1$, two non-overlapping q-grams must be queried. Choosing only one would miss results when the modification falls in the range of the q-

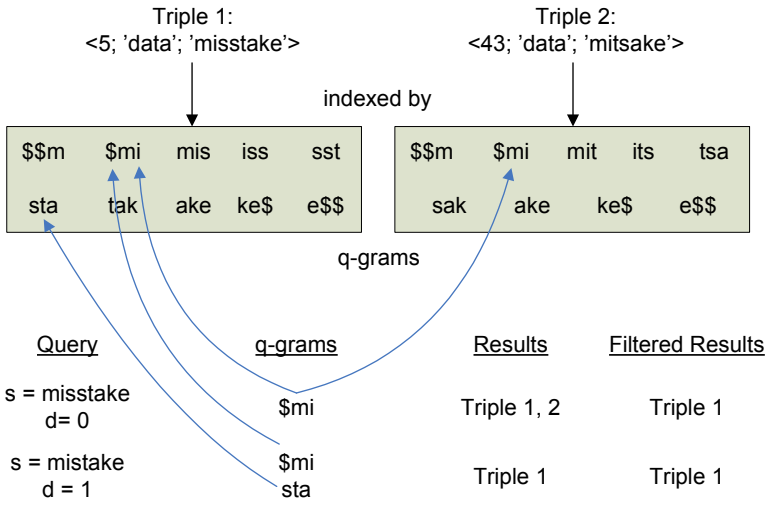

Figure 4.4: Similarity search using q-grams

gram. Two overlapping q-grams will miss results when the modification falls in the area of the overlap. Two non-overlapping ones will find all matches: the difference between the two strings might fall in the range of either one, but not both. Results must also be filtered for false positives. Looking up "mistake" with $d = 1$, q-grams "$mi" and "sta" can be used in conjunction and will match the first triple. In general, for a similarity search of distance d, $d+1$ q-grams must be queried. When not enough non-overlapping q-grams can be generated because the search string is too short, all overlapping q-grams are used. Not all results might be found in this case. The following q-gram indexes can be created:

Value Similarity Index This is comparable to the Values index presented above, but only triples containing string-typed values can be indexed. For a string of length l the triple is inserted $l + 2$ times at $h(q_i), 1 \leq i \leq l + 2$.

Attribute-Value Similarity Index Searches can be limited to the values of a particular attribute with this index. Triples are inserted at $h(att) \circ h(q_i), 1 \leq i \leq l + 2$.

Attribute Similarity Index Similarity searches on attributes (schema level) are needed for the applications targetted by UniStore (sec. 1.2). To accelerate them, q-grams are created for the attribute strings and triples are inserted at $h(q_i), 1 \leq i \leq l + 2$.

These indexes are used by implementations of $Extraction$ and combinations of $Extraction/Selection$. It must be noted that they consume a considerable amount of storage space, particularly for long strings. For n triples with an average indexing string

length of m, $n * (m + 2)$ copies of the triples must be stored ($O(n * m)$). The attribute similarity index can also cause a skewed storage load when there are not many distinct attributes. Normally, P-Grid is able to balance this with its sophisticated algorithms, but as the keys are identical in this case, this is not possible. It is possible to alleviate this by appending random bit strings before applying the hash function. A prefix query can still be used for retrieval. This was done in the tests in chapter 9.

Further details on q-gram indexes can be found in [Wie06, GIJ+01].

4.1.3 Remarks

Note that not all these indexes are required. Similar to relational databases, they can be used to optimize for certain access patterns. A minimum setup only using the AV index is possible. The OID index is only required by certain implementations of $Materialization$. The value index need only be created if queries on all values happen often. Of course, similarity indexes should only be created if string similarity queries are common. As noted above, indexing is done on a triple basis. In practice this will be used to specify indexes on an attribute or tuple basis. New indexes can also be added later.

All in all, this storage scheme supports the expected usage scenarios really well. The data is self-describing, no central catalog is necessary, which would be a bottleneck both on the physical and organizational level in scenarios where many users contribute data. The fine-grained granularity of the scheme makes it possible to extend existing relations both horizontally and vertically. Similarity indexes allow efficient querying based on Levenshtein edit distance. The main drawback is the increase in storage space. The annotations (OIDs and attributes) are duplicated for triples in the same row or column, and each index stores the complete triple data. The triples are also sent over the network during indexing and when the peer's paths change, increasing bandwidth consumption. Possible remedies are:

- Store the triples in one index only and reference them from all other indexes. For example, the OID index could be used as main index, with $OID \circ att$ as primary key. This approach requires additional lookups which can be very costly and make some optimizations impossible.
- The local storage used on each peer can be optimized to handle and eliminate some of the redundancies of the indexing scheme. For example, duplicate OIDs (for the OID index) and attributes (for the AV index) need to be stored only once,

thus restoring the relational structure of the data locally and possibly during transmission.

- P-Grid compresses messages which can reduce bandwidth for duplicated triple data considerably.

Note that storing a triple more than once is equivalent to replication, a desired feature in P2P networks, where peer failures are to be expected. A proper balance between limitations of storage and network capacities on one hand and efficiency of access and reliablity on the other must be found.

4.2 Querying Structured Triple Data

4.2.1 Vertical Query Language

The Vertical Query Language (VQL) [Sch06, KSHS06a] has been designed for querying triple data as seen in fig. 4.1, thus providing the frontend to query processing in UniStore. It is based on SPARQL [PS06], an RDF query language. The syntax looks like this:

```
SELECT [ DISTINCT ] <projection>
WHERE { <triple definition>
     [ FILTER <expression> ] }
ORDER BY <variable> [ ASC | DESC | NN <value> ]
[ LIMIT <n> ] [ OFFSET <m> ]
```

Queries can operate on all three components of a triple: OIDs and attributes (*schema level*) and values (*instance level*). In the following, available features will be illustrated with example queries. It is assumed that two relations `car` and `dealer` exist, with `car.dlrid` referencing `dealer.id`:

```
car(name, hp, price, dlrid)
dealer(id, dname, city)
car.dlrid → dealer.id
```

A basic query to display the price and horsepower of five BMW cars with at least 100 HP would look like this:

```
SELECT p, hp WHERE {
   <o; 'name'; 'BMW'> <o; 'price'; p> <o; 'hp'; hp>
   FILTER hp >= 100
} LIMIT 5;
```

In the `WHERE` clause triples are specified with the same syntax as explained in sec. 4.1, with OID, attribute and value components. Each component can either be a concrete value or a variable. The latter are used to establish correspondences between triples: "`o`" ensures that all triples have the same OID (i.e., belong to the same tuple). All tuples that match the specified criteria form the result set. Additionally, variables can be used to specify constraints other than exact-match with `FILTER` clauses. The available operations are $=$, $<$, $>$, $\leq$, $\geq$, $\neq$. For "fuzzy" searches, a similarity operator $\sim$ is available, with a numeric parameter specifying "how close" matches must be. For string data, this can be used in conjunction with the Levenshtein edit distance metric. Given two strings, it determines how many character insertions, deletions or substitutions are required at minimum to transform one string to the other. The following query searches for cars from dealers located in "Ilmenau" with some tolerance, so misspellings are found as well. "Ilemnau" (distance 2) will be located, but not "Illemnau" (distance 3). The query includes a join `car.dlrid = dealer.id`, specified implicitly by using the variable `did` in two triples referencing the two relations.

```
SELECT dn, da, n, p WHERE {
   <o; 'name'; n> <o; 'price'; p> <o; 'dlrid'; did>
   <d; 'id'; did> <d; 'dname'; dn> <d; 'city'; da>
   FILTER p < 10000
   FILTER da ~ 'Ilmenau', 2
};
```

An `ORDER BY`-clause is available and provides the functions $Minimum$ (`ASC`), $Maximum$ (`DESC`) and $Nearest\ Neighbor$ (`NN`). The last computes the distance of a triple component to a provided reference and sorts the tuples from minimum distance to maximum. Using this, the previous query can be modified to list "BMW" cars first, minor misspellings will be displayed afterwards. In conjunction with `LIMIT 5` this produces a *Top-N* query:

```
SELECT dn, da, n, p WHERE {
   <o; 'name'; n> <o; 'price'; p> <o; 'dlrid'; did>
   <d; 'id'; did> <d; 'dname'; dn> <d; 'city'; da>
   FILTER p < 10000
   FILTER da ∼ 'Ilmenau', 2
} ORDER BY n NN 'BMW', LIMIT 5;
```

All operators can also be applied at schema level. Suppose that not all car dealers committed themselves to the `car` schema and used `dlr` or `iddlr` instead of `dlrid` as the last attribute. The previous query can easily be modified to account for this.

```
SELECT dn, da, n, p WHERE {
   <o; 'name'; n> <o; 'price'; p> <o; dlratt; did>
   <d; 'id'; did> <d; 'dname'; dn> <d; 'city'; da>
   FILTER p < 10000
   FILTER da ∼ 'Ilmenau', 2
   FILTER dlratt ∼ 'dlr', 2
} ORDER BY n NN 'BMW', LIMIT 5;
```

Similarity joins on instance and schema level with a user-defined distance are also possible. Supported data types are string, integer and float. In this work, only string and integer will be used.

4.2.2 VQL Algebra

In [Sch06], a parser and an algebra for VQL are presented as well. The algebra has much in common with its relational counterpart. The parser dissects a VQL query and determines what operations of the algebra must be applied in what order to compute the result. As the algebra is closed, results from operations can be used as input for other operations. The output of the parser is a tree consisting of the algebra operations ("logical operations") with the last operator to be processed at the root and data flowing from the leaves to the root. This is the logical operator plan [Kos00].

The next section documents the available operations. Their input is x. Informally, each x is an ordered list of "jtuples". A jtuple stores triple data in a structured way. A more formal definition will be given afterwards. Logical operators are usually typeset in $italics$ in this thesis.

$Extraction$

$\xi_p(x)$

All triples conforming to predicate p are extracted from the DHT; for example, all triples with a given attribute or simply all triples.

$Materialization$

$\omega_{att_1,...,att_n}(x)$

This operation extends the input jtuples vertically by the given attributes. For each attribute specified, an additional triple is appended to each jtuple in the input list.

$Selection$

$\sigma_p(x)$

Just like in the relational algebra, all jtuples are filtered according to the predicate. All available operations ($=$, $<$, ...) are supported both on schema and instance level.

$Join$

$\bowtie_p (x_1, x_2)$

The cross product of both inputs is filtered by the predicate. Similar to $Selection$, all available operations are supported on instance and schema level. Multiple predicates can be specified, they are applied conjunctively.

Ranking

$\varphi_{f_1,...f_n,limit,offset}(x)$

Three ranking functions are supported: $Minimum$, $Maximum$ and $Nearest\ Neighbor$. When multiple functions are specified they are applied in a nested fashion (see sec. 7.2 for details). The same function can be used multiple times for different variables. This operation also handles the `LIMIT` and `OFFSET` clauses.

Projection

$\pi_{att_1,...,att_n}(x)$

Just like in the relational algebra, the projection eliminates all variables not present in the parameter list. In VQL, variables can refer to OIDs, attributes or values. The output is a tabular structure comparable to the result generated by a relational database. Each variable specified in the `SELECT` clause is output in its own column, for every jtuple ("row") of the result set. This output is not part of the algebra. As no further processing will take place, this is not required.

The operators presented will most likely be extended in the future. For example, skyline or aggregation queries could be supported. A proposal for the latter is presented in sec. 7.6.1.

Definition of Operator Input/Output x

x represents a list of jtuples, which store triple data in a structured way. It is not sufficient to just use lists of triples as input/output of operators. For example, $Materialization$ makes use of the relationships between triples and matches them on the OID to form tuples, which must be reflected in the output. This is abstracted in the following.

The simple case where each operator only operates on and produces one result row is considered first. One triple (OID, att, val) is the result of a simple $Extraction$ operation and can be represented as $t = (o, a, v)$. Each $Materialization$ results in a number of additional attribute/value pairs, extending this triple to a generalized tuple form:

$u = (o, a_1, v_1, ..., a_m, v_m), m \geq 1$, which can be used for triples as well ($m = 1$). $Joins$ will operate on two such tuples, comparing a component from the first with a component from the second. Note that the result of the $Join$ must preserve both tuples, because later operations can take place on either one. Therefore, another tuple is used to store the tuples from the left and the right side, called "jtuple" (join tuple). Each $Join$ simply adds a new tuple to the jtuple, which results in output of the form $v = (u_1, ..., u_n), n \geq 1$ for $n - 1$ processed $Joins$.

In the general case, each operator processes an ordered list $x = (v_1, ..., v_l), l \geq 0$ of jtuples (ordered because the order established by $Ranking$ must be preserved). Using this definition, a triple as well as all kinds of intermediate results can be expressed by x and used as input and output of all operations. A structure to store and exchange this kind of data must be implemented as part of the query processor (see sec. 7.1).

4.2.3 Example: Logical Plan and Parameter Representation

The VQL parser produces the logical plan from a query. The plan is optimized – for example, $Materializations$ are inserted directly before operators depending on them, $Selections$ are pushed down to the leaves of the tree, thus reducing the size of intermediate results[4]. Consider this query:

```
SELECT o, n WHERE {
   <o; 'name'; n> <o; dlratt; did>
   <d; 'id'; did> <d; 'city'; dc>
   FILTER dlratt ~ 'dlr', 2
} ORDER BY n NN 'BMW';
```

The *text representation* of the logical query plan generated by the parser for this query is shown next. The structure of the plan is depicted in fig. 4.5.

[4] in database literature, algebra optimizations are carried out by an additional rewriter component; this distinction is not made here

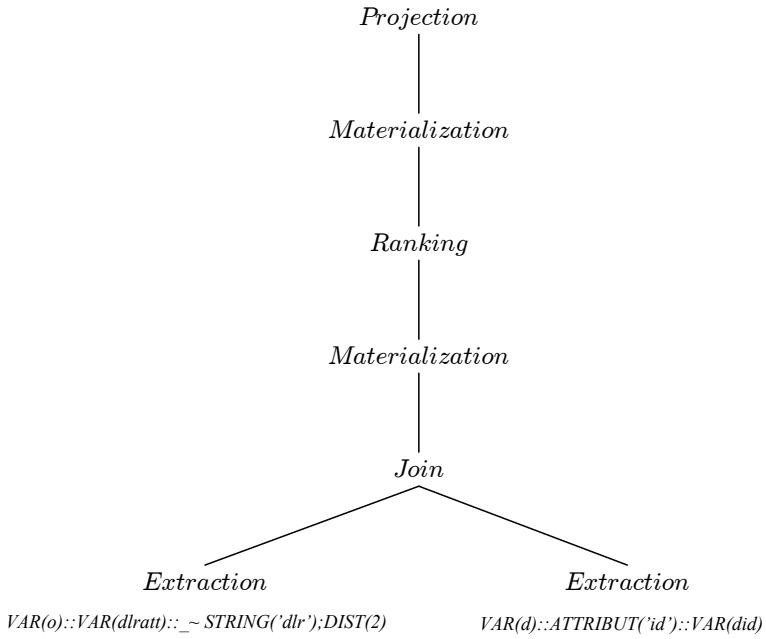

Figure 4.5: The structure of the logical query plan

```
Projection<VAR(o)::_::_;VAR(o)::ATTRIBUT('name')::VAR(n)>(
  Materialization<VAR(d)::ATTRIBUT('city')::VAR(dc)>(
    Ranking<NN;VAR(o)::ATTRIBUT('name')::VAR(n);STRING('BMW')>(
      Materialization<VAR(o)::ATTRIBUT('name')::VAR(n)>(
        Join<VAR(o)::VAR(dlratt)::VAR(did) =
             VAR(d)::ATTRIBUT('id')::VAR(did)>(
          Extraction<VAR(o)::VAR(dlratt)::_ ~ STRING('dlr');DIST(2)>(_)),
          Extraction<VAR(d)::ATTRIBUT('id')::VAR(did)>(_)))))
```

The parameters of each VQL operator, enclosed in angle brackets ($< / >$), are given in a special notation which must be parsed for further processing. They provide information about:

- the scope of an operation: OID, attribute or value
- variable names and concrete values used in the query
- type information
- arguments for comparison and distances

As explained in sec. 4.2.2, each jtuple stores many components (which can be OIDs, attributes and values) linked together in certain ways. Therefore, the most important information is what component of each jtuple should be processed by the operator. This is specified by a triple-like notation. Consider the left child operator of the join, an *Extraction* with the parameter `VAR(d)::ATTRIBUT('id')::VAR(did)`, corresponding to the triple `<d; 'id'; did>` in the query. In the OID part, `VAR` indicates that a variable was used and its name `d` is given. `ATTRIBUT` specifies a concrete attribute. The value part is similar to the object part. This operation will fetch all triples with attribute `id`.

The second *Extraction* `VAR(o)::VAR(dlratt)::_` $\sim$ `STRING('dlr');DIST(2)` can be read from left to right as "for the component `VAR(o)::VAR(dlratt)::_`, perform a similarity ($\sim$) comparision against string `dlr` with distance 2". But what component is addressed here? No concrete attribute name is given, and the value part contains only "`_`". This indicates a schema level operation on the attribute itself – all triples with an attribute similar to `dlr` will be fetched. Next, the *Join* ensures that the values of the fetched triples correspond. Notice how the notation is used to address the components needed: the object part specifies the left and right branches of the *Join*, respectively. The attribute for the right side is given, for the left side it is not known, so the variable is provided instead. The result of the join will be a list of jtuples, each consisting of two tuples (i.e., the data fetched by the *Extractions*). `OMEGA<VAR(o)::ATTRIBUT('name')::VAR(n)>` materializes the `name` attribute. For that, a OID must be available. Notice that there are two, one for each of the *Extractions*. The OID component of the notation, `VAR(o)`, identifies the correct one to use. Finally, an operation on an OID can be seen in the *Projection*, which outputs the OIDs associated with variable `o`, specified as `VAR(o)::_::_`.

The logical query plan must be analyzed in this way by the query processor and the collected information used in the subsequent stages. This is discussed in sec. 5.3. Further details on the syntax and semantics of VQL parameters can be found in [Sch06].

5 CouPé: A Query Processor for UniStore

The first section highlights the challenges for query processing in UniStore along with possible solutions. A query processor architecture is presented in section two and one component – the query planner, responsible for mapping logical to physical operators – introduced in the last section.

5.1 Challenges

Traditional distributed databases share many of the goals of DHT databases and consequently, many of the challenges in query processing are the same:

Data Localization All data relevant to the query must be found, and all corresponding nodes contacted. Data can be replicated to enhance parallelism, fault tolerance and load balancing, which makes the choice of the nodes to utilize harder.

Efficient Distributed Processing Dependent on data distribution, node capabilities and available resources, the query must be executed in a distributed setting efficiently, with certain optimization goals (for example, minimum response time). Parallelism and specialized operators, like semi joins and hashfilter joins, should be used where possible.

Account for Communication Costs They often are the dominating factor in distributed settings. Important parameters for query planning are the available bandwidth, latency of links and the size of messages. In many cases it is best to process data on the nodes storing it, reducing the cardinality and therefore the transmission costs for subsequent processing. On the other hand, this might lead to hotspots forming on nodes with popular data. Parallelizing query processing for speedup causes higher communication costs – a balance must be found.

Global knowledge in the form of schemas and statistics makes it relatively easy to deal with these challenges in distributed databases. The execution of a query in such a

system progresses as follows [Sat06]:

Query Transformation The query[1], posed on the global schema of the database, and submitted to a central query processor, is parsed and transformed to the equivalent algebra representation, which may also be optimized.

Data Localization Locate all data relevant to the query by consulting the distribution schema. All nodes storing data must be involved in the processing of the query (except for replication nodes).

Global Optimization Based on parameters like execution and communication costs and global statistics, an optimal query plan is generated which specifies where and when to execute operators. The participating nodes receive the parts of the plan they are responsible for.

Local Optimization and Code Generation The subplans are optimized locally and executable subplans are produced.

Execution The subplans are executed, with the central processor overseeing execution, receiving intermediate results and producing the final answer.

In contrast, DHT databases can only rely on local knowledge. Only the hash functions which map application keys to DHT keys (sec. 4.1) can be considered global knowledge and are used for efficient data localization. However, they do not provide as much information as the distribution schema in distributed databases: they do not reveal on how many or on which peers data is stored, making it practically impossible to generate a good query execution plan upfront. Thus, a mechanism for generating and refining a physical query plan while the query is running is desired – the emphasis should be on local optimizations ("adaptive query processing").

UniStore also does not enforce a global schema – users are encouraged to extend existing data as needed. It is also expected that data is inconsistent both on instance level (for example, typing errors) and schema level (different schemas for the same concepts). Therefore, similarity and schema level operators are needed. The algebra presented in sec. 4.2.2 already provides them, they must be implemented efficiently. Operators in general should not fail in the face of adversity, but instead provide best effort. For example, missing data or mismatching data types should not cause failure of the query, the malformed data should simply be ignored.

Best effort is also a requirement for the query processor as a whole: peers can join and leave the network at any time, even fail without warning, but the system may not

[1] formulated in SQL, for example

be disrupted by this. While malicious nodes are also a concern in P2P systems, they are not considered here. Due to these cirumstances, 100% complete results are not a realistic goal in such a system. Luckily, because of the sheer amount of data, this is not necessary: often there are many equally good results, and providing a few of them fast is more important than delivering all. This point is also illustrated by mainstream search engines. Therefore, fast initial responses which can be refined later are desired.

Features of P-Grid such as prefix queries should be used to enhance the system. While these features might not be available in other DHTs it is often possible to emulate them.

The design presented in the remainder of this chapter deals with all of these challenges.

5.2 Overview of Query Processing in UniStore

In fig. 5.1, the processing of a query in UniStore is depicted. Each component exists on every peer. First, the declarative VQL query issued by a user or an application is fed into the VQL parser, which generates an optimized logical operator tree as outlined in sec. 4.2.3. Next, the query planner substitutes appropriate implementations – physical operators – for each logical operator, yielding a physical query plan. Multiple planner implementations are available, differing in the physical operators they substitute. Most of them will only change part of the plan, but it must be ensured that all logical operations result in executable code, so more than one can be applied. As global knowledge is not available in P2P systems, it is difficult for the initiating peer to generate an optimal physical query plan. Therefore, the plan passed to the execution engine may still contain logical operators which are substituted during processing with a “lazy evaluation” approach, thus making it possible to use local knowledge available at the peers involved. Of course, the planners to use for this must be included in this case. The execution engine evaluates the operators in the correct order, they in turn access the data stored on peers or process data provided by other operators. The final result is returned to the user on the initiating peer.

Note that while the VQL parser will be invoked on the initiating peer only, the planner, engine and operators of many peers are utilized. As the parser has already been implemented as part of a diploma thesis [Sch06], this work only covers the planners, execution engine and operators, with the focus on the last two. The query planners only need to be able to alter plans to test different operators and provide relatively sim-

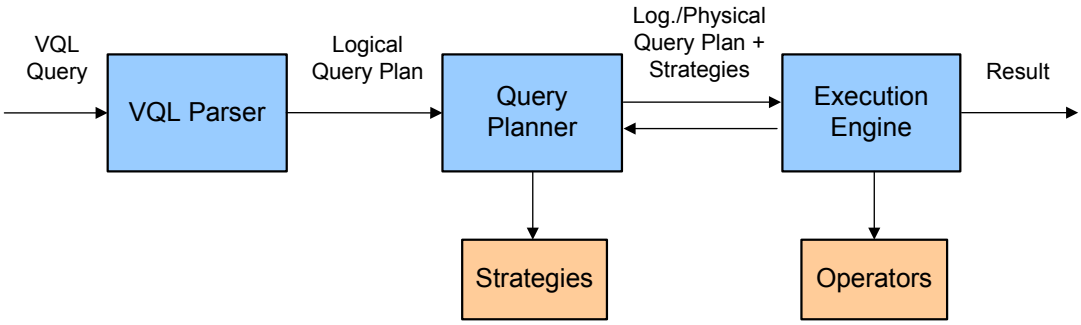

Figure 5.1: Query processor architecture

ple optimizations. Complex optimizations, for example involving data distributions and selectivity, are left for future work. They are discussed in the following section. The execution engine and the operators are documented in chapters 6 and 7.

5.3 Query Planners

The planners are responsible for instantiating logical operators, supplied as part of logical query plans by the VQL parser, with physical operators. An $m : n$ mapping is used: m logical operators can be implemented by n physical operators ($m, n \geq 1$). Each planner is able to deal with a subset of logical operators and more than one can be applied to a plan. A basic planner is available which can map all logical operators to implementations. Table 7.1 shows all planners in conjunction with the physical operators they substitute. Planners can be invoked at two points in the query execution process:

- Instantiation can happen before the query is passed to the execution engine for the first time. Once an operator has been mapped, other planners have no effect on it. As this happens on the query initiator, the resulting query plan may not be optimal.
- During query processing, any remaining logical operators are mapped by the planners as needed (“lazy instantiation”). This can incorporate local knowledge like current load or data distributions and opens the door for advanced optimizations, an area where much research is needed in the context of P2P systems (“adaptive query processing”). The planners to be used must be specified in the query plan. While the current implementation is only able to reference one it can easily be

extended if necessary.

Since complex optimizations will not be performed, the second approach is not needed. Therefore, the logical query plan is converted to a physical plan on the initiator. Each logical operation in the plan has a type (i.e., $Extraction$, $Materialization$) and parameters. This information must be obtained by parsing the query plan emitted by the VQL parser (sec. 4.2.3).

For each logical operator in the plan the query planners are consulted until one claims responsibility. It parses the parameters of the operator, initializes physical operators with them and places them in the plan. When each logical operator has been handled, this results in a plan consisting of physical operators only. A planner can also consume multiple logical operators at once, as mentioned above. Finally, the physical plan is passed to the execution engine, which will be discussed next.

6 Execution Engine

In this chapter a key component of the query processor, the execution engine, will be presented. The first section discusses possible plan processing strategies. The Mutant Query Plan (MQP) strategy is chosen and section 2 and 3 explain its implementation in CouPé. Extensions for it are presented in section 4. After that, ways to keep the user informed about the state and completion of a query are discussed. The final section provides a short summary and outlook.

6.1 Execution Strategies

Given a query plan consisting of logical and/or physical operators[1], the query processor must find a way to arrive at the results. One part of this process, the mapping of logical to physical operators, has been discussed in sec. 5.3[2]. The remaining task is the evaluation of the physical plan itself utilizing the operator implementations. Possible strategies for this are:

Data Shipping This simple approach fetches all data and processes the query locally. This incurs high communication costs and parallelism is non-existent. In cases where the same data set is queried frequently and can be cached this approach is a viable solution.

Query Shipping The query (or a subplan) is passed to the peer storing the data, processed and the results are sent back. For most operations, communication costs will be lower relative to data shipping and higher concurrency can be achieved. This approach is used in most distributed databases.

Hybrid Shipping None of the previous approaches are ideal for all queries. Hybrid shipping chooses the better strategy depending on the query and the data already

[1] an example plan consisting of logical operators is depicted in fig. 4.5

[2] although this will be done on the initiator in this work and not during processing, the latter will also be discussed in this chapter, as the engine already supports it

cached at the client. This requires a more complex query planner.

Details on the aforementioned shipping methods can be found in [Kos00].

Mutant Query Plans (MQPs) [PM02b] This concept has been designed for systems where a central processing component is not feasible. In addition to the operator tree, query plans also contain data and references to data. The references are resolved by routing to the nodes storing the data. Operators are evaluated if possible, they store results in the plan. In the basic approach this happens sequentially. Thus, each peer *mutates* the plan and forwards it. When all operators have been processed, the plan, now containing the results, is sent back to the inititator. As both query and data are shipped in this approach it is also termed combined shipping. The lack of a central coordinator makes it a good fit for CouPé. Another interesting aspect is the low footprint. Since a plan is processed and then forwarded, it only consumes resources on one peer at a time[3] – the complete state of the query is encapsulated in the plan. In query shipping systems, resources on multiple nodes are occupied at the same time. The P2P environment can be fragile and network connections can time out – the fewer resources allocated during this idle time, the better. On the flip side, this basic variant does not support concurrency. The authors propose *strains*, which are in effect parallel MQPs but require some kind of synchronization at a later point, again tying up resources. As the whole query plan is available on each peer many optimizations become possible – it can be restructured and optimized with the aid of local knowledge.

Because of these traits, MQPs will be used as the basic concept for plan execution. Their simplicity allows for a quick initial implementation, while parallelism and other enhancements can be added later.

6.2 MQPs in CouPé

The structure of an MQP as used in CouPé is shown in fig. 6.1. It is similar to the plan generated by the VQL parser depicted in fig. 4.5, but in addition to the logical operators (typeset in *italics*) it also contains physical operators (typeset in `monospace`), `Extract` and `LocalJoin`, two implementations of *Extraction* and *Join*[4]. Physical operators are state machines and perform tasks like initialization, routing of the plan or plain data processing. They can extract data from the DHT or consume data provided by their children in the plan. Each physical operator is able to store the data it produces and in

[3] except for the short moment of transmission

[4] described in detail in sec. 7

turn make it available to its parent operator. Therefore, data flows from the leaves to the root of the operator tree. For transport between peers an MQP, including all of its data and operator state, is serialized and attached to a special P-Grid message.

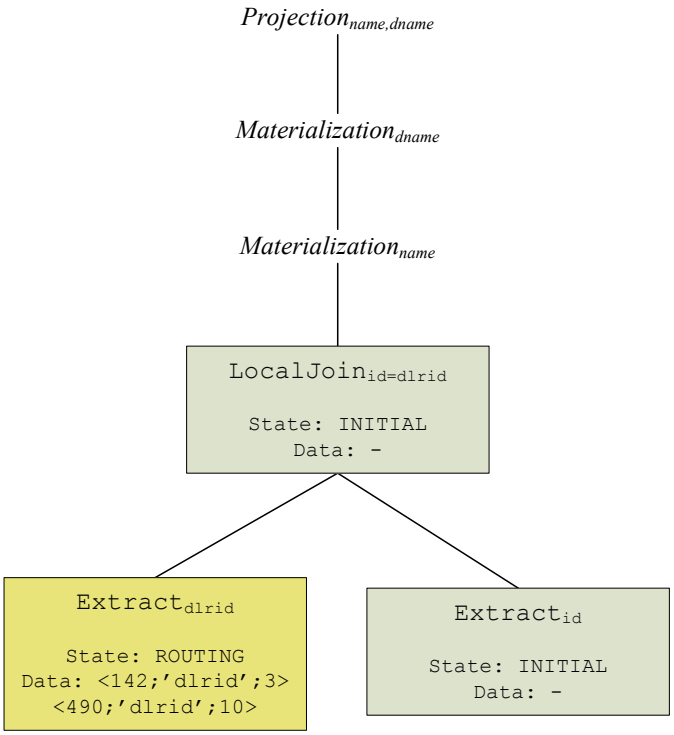

Figure 6.1: A Mutant Query Plan in CouPé

This work enhances the original MQP proposal presented in [PM02b]. For example, operator implementations are relatively autonomous – they can duplicate the query plan and even change its structure and the data stored, adding another *mutation* dimension to the concept. Therefore, the term M^2QP will be used in the following to distinguish it from the original MQP proposal. For simplicity, "query plan" and "plan" are used as well and also refer to M^{2}QPs.

6.3 Serial M^{2}QP Execution

First, a straightforward implementation of the concept described above will be presented. As it features no parallelism and processes the operators in sequence it is termed “serial”. Other execution strategies base on it, so it will be examined in detail.

When a query plan has been submitted to the execution engine, it must make sure that operators are started in the correct order: child operators must run before their parent so that the data the parent relies on is available when it is run. This can easily be achieved by traversing the operator tree in postorder, which generates a list of the operators (“operator queue”) with the parent guaranteed to appear after all of its children. Left subtrees are evaluated before right subtrees. Fig. 6.2 shows the queue for the plan in fig. 6.1. The root operator which is responsible for storing the final result is at the bottom of this list. Note that this execution order also fits the logical plans generated by the VQL parser, which makes sure that $Materializations$ are only inserted immediately before the operators depending on them and $Selections$ are pushed down to the leaves [Sch06]. Both measures reduce the amount of data stored in the operators at a given time. This is vital for the M^{2}QP concept as the complete plan is sent around the network.

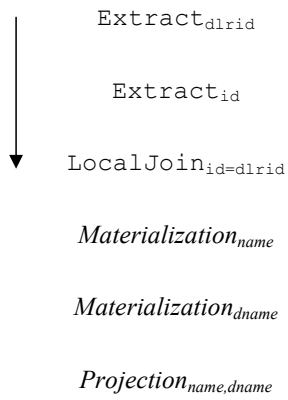

Figure 6.2: The corresponding operator queue

The central algorithm for processing an M^{2}QP is shown in fig. 6.3 and will now be discussed in detail. First, the root operator is checked for completion. In this case the result is sent to the initiator and execution is complete. Otherwise, the engine checks the queue for operators to process. The first physical operator encountered

```
void processPlan(Plan p) {
    while (true) {
        if (p.rootOp.isDone()) {
            sendResult(p.getInitiator(), p.rootOp.getResult())
            break
        }
        Operator[] opQueue = p.postOrder()
        Operator activeOp = findFirstNotDone(opQueue)
        if (activeOp is LogicalOperator) {
            // replace with physical operator
            QueryPlanner.instantiate(p, activeOp)
            continue
        } else {
            // pass control to operator for processing
            boolean doRoute = activeOp.process()
            if (activeOp.isDone()) {
                activeOp.removeChildren()
            }
            if (doRoute) {
                // send plan to next peer, finished on this one
                routePlan(p, activeOp.getDestination())
                break
            }
        }
    }
}
```

Figure 6.3: Serial postorder processing of an M^{2}QP

which is not marked DONE or the first logical operator is considered. In the latter case, the query planner is consulted for the corresponding physical operator(s). It will insert them in the plan, so the queue must be updated and the process starts over (with a physical operator now being available). Otherwise, the execution engine passes control to the operator which performs tasks depending on its current state. With the return value it specifies whether the plan must be routed before processing can continue. The operator can also mark itself as DONE. When routing is required, the engine serializes the plan and sends it to the peer specified by the operator via its P-Grid path, using P-Grid's routing layer. On the receiving peer, the plan is passed to the local execution engine which starts over with this algorithm[5]. Once an operator has moved to the DONE state the engine will move to the next one in the operator queue. The children of the operator can be removed at this point, which reduces the size of the plan. The algorithm processes as much of the plan as possible – until the result is available or an operator requires routing. This means that a plan where no operator requests routing will be processed on the initiator only. A completed plan is shown in fig. 6.4. Failures are tolerated as far as possible ("best effort"). If a fatal error occurs, a failure notice is sent to the query initiator.

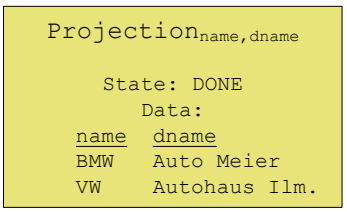

Figure 6.4: The final plan

6.3.1 Forward Processing

One drawback of this approach is the sometimes unnecessary size of the plans generated during processing. Consider a query which first extracts a big amount of data stored across many peers and then performs a selection on it. The physical operators `Extract` and `LocalSelection` can be used for this (discussed in detail in sec. 7). Following the serial strategy, `Extract` will be executed first, traversing the peers and accumulating all data, before `LocalSelection` is run and gets a chance to reduce the number of jtuples. To improve this, the concept of *forward processing* is introduced: operators can be "chained" to their parent operators, and each time an operator returns

[5] unless the active operator has marked itself as DONE, execution will resume with it

```
...
boolean doRoute = activeOp.process()
// additional forward processing code
Operator op = activeOp
while (op.processParent()) {
    Operator parent = op.getParent()
    parent.process()
    op.clearJTuples()
    op = parent
}
// end additional forward processing code
if (activeOp.isDone()) {
    ...
```

Figure 6.5: Forward processing algorithm

from its `process()` method, the engine immediately executes its parent operator, even if the current one is not yet in the DONE state. In the example above, `LocalSelection` will be called after `Extract` has extracted data on the first peer. It processes the jtuples supplied by `Extract` and stores the matching ones. Afterwards, the execution engine clears the jtuples in `Extract`. Therefore, the plan size is reduced by the number of filtered jtuples. An arbitrary number of operators can be linked together in this fashion, fig. 6.5 shows the algorithm which extends the `processPlan()` method shown in fig. 6.3.

Forward processing also allows an easier implementation of physical operators. Some logical operators have common functionality[6]. Instead of implementing it in two physical operators it can be isolated to one. The distinct functionality is implemented in additional ones which will be used together with the common operator and forward processing. This reduces code duplication and the chained operators still have the same properties as a single one as far as the execution engine is concerned. It is also possible to run the chained operators in parallel (pipelined parallelism [YM98]).

A great disadvantage of serial M^{2}QP execution is slow processing, as no multi-peer parallelism is used. Approaches to improve this will be presented next.

[6] for example, the VQL *Extraction* operation also allows similarity *Selection* on schema level

6.4 Parallel M^{2}QP Execution

One form of parallelism, pipelined parallelism, has already been mentioned in the previous section, but it is limited to one host. The advantage of a large DHT database is that processing can happen on many nodes at the same time. Two forms of parallelism for this are [YM98]:

Intra-operator Parallelism One operator is simultaneously processed on multiple peers. One example is the $Extraction$ of data from an index distributed over many peers. Because there are no interdependencies, the operator can be executed in parallel. It is necessary to merge the data at some point to generate the final result. Other operations which can profit from this approach are $Materialization$, $Selection$, $Join$ and $Projection$.

Inter-operator Parallelism Operators or subplans are run in parallel. In contrast to pipelined parallelism no communication takes place – the subplans must be independent of eachother so that no plan requires the output of any other. The results are later combined. This approach will be used to process branches of binary operators in parallel.

These forms of parallelism have been implemented in the execution engine and the physical operators. Only the engine will be discussed in the next three sections, but some physical operators will also be introduced (typeset in `monospace`) . A more detailed description of them can be found in sec. 7.

6.4.1 Prefix Queries

One drawback of serial M^{2}QP execution is the high latency, as the peers storing data required by an operator are contacted in sequence. They are addressed by their P-Grid path in this case. However, P-Grid can also send messages to groups of peers specified by their common path prefix using prefix queries (sec. 2.2). This can be used in conjunction with the AV index presented in sec. 4.1.1: an operator requiring the triple data of attribute x simply addresses the plan to prefix $h(x)$ and P-Grid routes it to all responsible peers, where the plans are processed in parallel and the operator extracts the locally stored data (intra-operator parallelism). `ParallelExtract` (sec. 7.3.2) and `ParallelAVMaterialize` (sec. 7.3.2) use this feature.

For an example, consider fig. 6.6. This is the same query plan as in fig. 6.1, but all

logical operators have been instantiated and prefix queryimplementations have been used. One $Materialization$ has been omitted for clarity and operator states and data are not shown. Processing starts at $\texttt{ParallelExtract}_{\texttt{dlrid}}$. The operator requests that the plan is routed to prefix $h(dlrid)$, so it is sent to all k peers storing data of the AV index for this attribute in parallel (AV_{dlrid}). Each one processes the plan just like in the serial case. Therefore, $\texttt{ParallelExtract}_{\texttt{id}}$ is processed next, again duplicating the plan and routing it to the l peers storing AV_{id}. At this point, $k*l$ plans exist in the network. These peers also process `LocalJoin` using the data stored in the two `ParallelExtract` operators. Note that each possible combination of AV_{dlrid} - AV_{id} fragments is generated by this query plan dissemination, thus ensuring correct join processing. One last time the plans are duplicated by $\texttt{ParallelAVMaterialize}_{\texttt{name}}$ and sent to m peers. The final `Projection` is processed locally on them and $k * l * m$ result plans are returned to the query initiator. Fig. 6.7 provides an overview: the peers associated with the AV indexes, the query initiator and the query plans sent between them are depicted for $k = m = 2$, $l = 3$ and 12 results are generated. The operators are associated with the peers on which they perform their main task (i.e., data extraction or processing, not routing). When multiple messages are transmitted between peers this is indicated by a number.

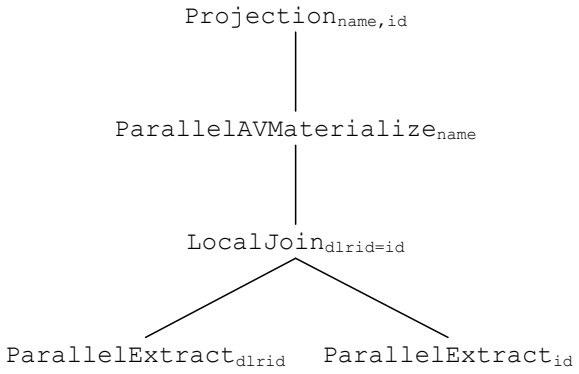

Figure 6.6: A physical query plan using prefix queries

This scheme has overhead. For example, two peers process `ParallelAVMaterialize`. Only one can process any given jtuple generated by the preceding operator, because the AV index for the attribute is split between the peers, but the prefix query mechanism sends the complete jtuple data to both. While the operator will discard data it cannot materialize, this still causes twice as much traffic as necessary.

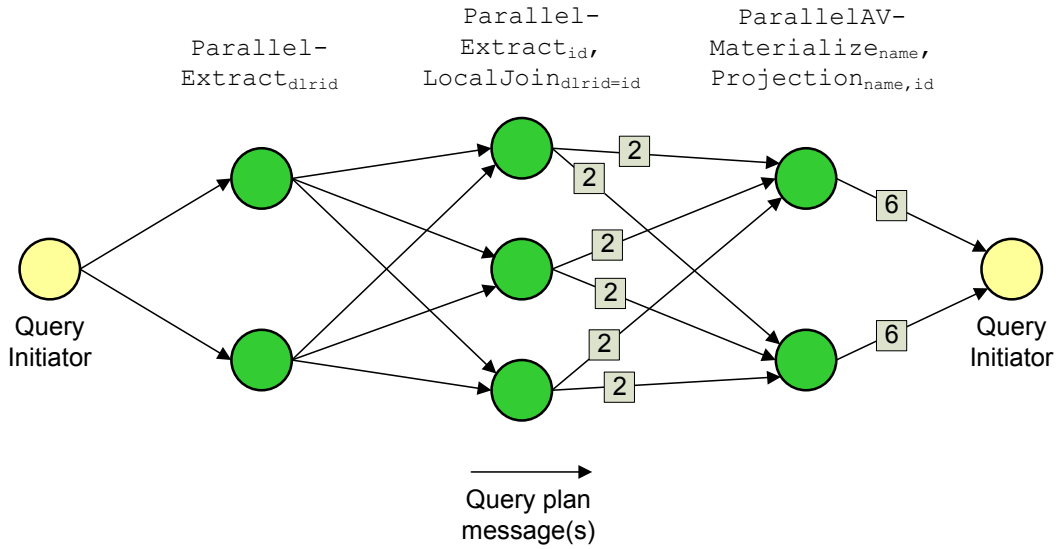

Figure 6.7: The messages sent during processing of the query plan

Plan Synchronization

The concurrent plans resulting from prefix queries must be merged. In the above example this happens on the query initiator: arriving result plans are merged with those previously received and the updated result is signalled to the application/user. This *online processing* [HHW97] strategy provides quick initial results, a requirement stated in sec. 5.1. After a timeout without new results the allocated resources are freed and the query is considered complete.

However, for operators which depend on the complete output of their children[7], synchronization during plan processing is required – all parallel M^{2}QPs with input for that operator must be collected and merged on one peer. But it is not even known how many parallel M^{2}QPs exist. One solution is to wait some time for plans, but this introduces an unacceptable delay in processing, especially with multiple blocking operators in a plan. Currently the only blocking operator in VQL is $Ranking$ and it will only appear once in a plan, but extensions of the VQL algebra might change this. In most cases it appears at the top of the plan, immediately below $Projection$ and $Materializations$ of jtuples with no dependency on it (fig. 6.8). As synchronization has to happen on the query initiator anyway, the plan could be rewritten so that $Ranking$ always appears below $Projection$

[7] called *blocking* operators because they stall pipelined processing

(fig. 6.9). The $Ranking$ implementation would route the plan to the initiator, where the final two operators are processed locally. However, $Ranking$ often reduces the number of jtuples in the plan by a considerable amount, for example when the `LIMIT` clause is used. A rewritten plan would materialize *all* jtuples, leading to higher bandwidth consumption and slower processing.

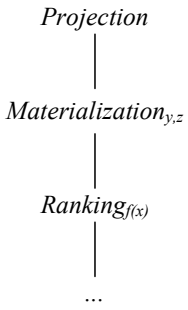

Figure 6.8: A typical VQL plan containing $Ranking$

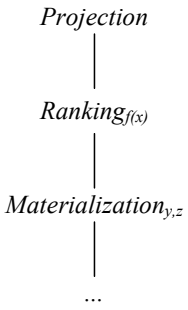

Figure 6.9: Rewritten plan to process $Ranking$ on the initiator

Because of this and to be able to handle future blocking operators like aggregation as well, a more general solution has been implemented. Each blocking operator routes the plan to a *synchronization peer*, where the jtuples stored in the child operators are added to those collected on the peer from other plans of the query. The accumulated jtuples are inserted in the plan and processing continues. This way, multiple *revisions* are produced, each getting more accurate. The plan is tagged with an incrementing revision number so the query inititator can identify the latest plan. In contrast to the aforementioned solution synchronization is not limited to the initiator. The plan can be split again afterwards, allowing for multiple blocking operators in one plan. After a timeout without any accesses the central peer discards the accumulated jtuples. Fig. 6.10 shows the extension of the original M^{2}QP algorithm of fig. 6.3 with synchronization. The

```
        ...
        QueryPlanner.instantiate(p, activeOp)
        continue
    } else {
        if (activeOp.isBlocking()) {
            GUID g = activeOp.getGUID()
            // get synchronization data for operator
            SynchData d = synchData.get(g)
            Operator leftChild = activeOp.getLeftChild()
            Operator rightChild = activeOp.getRightChild()
            // add jtuples from plan to synchronization jtuples
            d.addLeftJTuples(leftChild)
            d.addRightJTuples(rightChild)
            // put synchronization jtuples in plan
            leftChild.setJTuples(d.getLeftJTuples())
            rightChild.setJTuples(d.getRightJTuples())
            d.revision++
            p.revision = d.revision
        }
        // continue processing on synchronized jtuples
        boolean doRoute = activeOp.process()
        ...
```

Figure 6.10: Algorithm for synchronization

blocking operator is responsible for requesting routing to the synchronization peer and may only return `true` for `isBlocking()` when it has been routed to the peer.

Because all operators in the plan on top of $Ranking$ are processed for each generated revision, the traffic is multiplied by the number of plans received at the synchronization peer. This can be reduced by only generating a new revision for every n plans or by waiting a certain time between revisions.

Naturally, the synchronization peer would be specified by its path. Due to P-Grid's replication feature this is not possible – multiple peers with the same path can exist and a plan can arrive at any of them. To work around this, the plan is sent to a concrete peer specified by IP address and port. For every query a different peer can be chosen to balance load. When no IP address/port number is known, P-Grid's peer lookup can be used on a random binary P-Grid key to obtain the address of one responsible peer.

The single synchronization peer can become a bottleneck in big networks or for big query plans. For certain operations it is possible to use multiple peers and balance

load. In this work, only the general solution as outlined above has been implemented; optimized approaches are left for future work.

6.4.2 Plan Cloning

Prefix query parallelism relies on P-Grid's routing layer for plan duplication so it can only be used when appropriate indexes are available. Plan cloning eliminates this limitation by allowing operators to generate multiple copies of the current plan. Each one can be modified and routed to a different peer. This second approach to intra-operator parallelism provides great control over query processing.

One problem with prefix query operators was the increase in traffic caused by their "brute-force" multicasting, as shown for `ParallelAVMaterialize` in sec. 6.4.1. `ParallelOIDMaterialize` is an implementation using plan cloning which does not suffer from this problem. An example query plan and the exchange of plans during processing are shown in fig. 6.11 and fig. 6.12. First, the plan is routed to the two `Parallel−Extract` peers (prefix query). Each peer stores two jtuples, which are fetched, and for each one a separate plan is generated and routed to the corresponding peer of the OID index for $Materialization$. Therefore, each jtuple is only sent to a peer where it can be processed. The drawback is the overhead of the message headers and plan data structures, so the operator might only be efficient for relatively large jtuples. Four result plans arrive at the initiator.

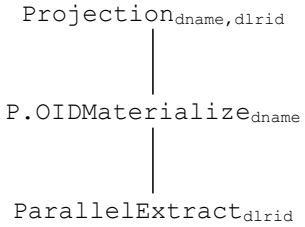

Figure 6.11: Plan cloning: example query plan

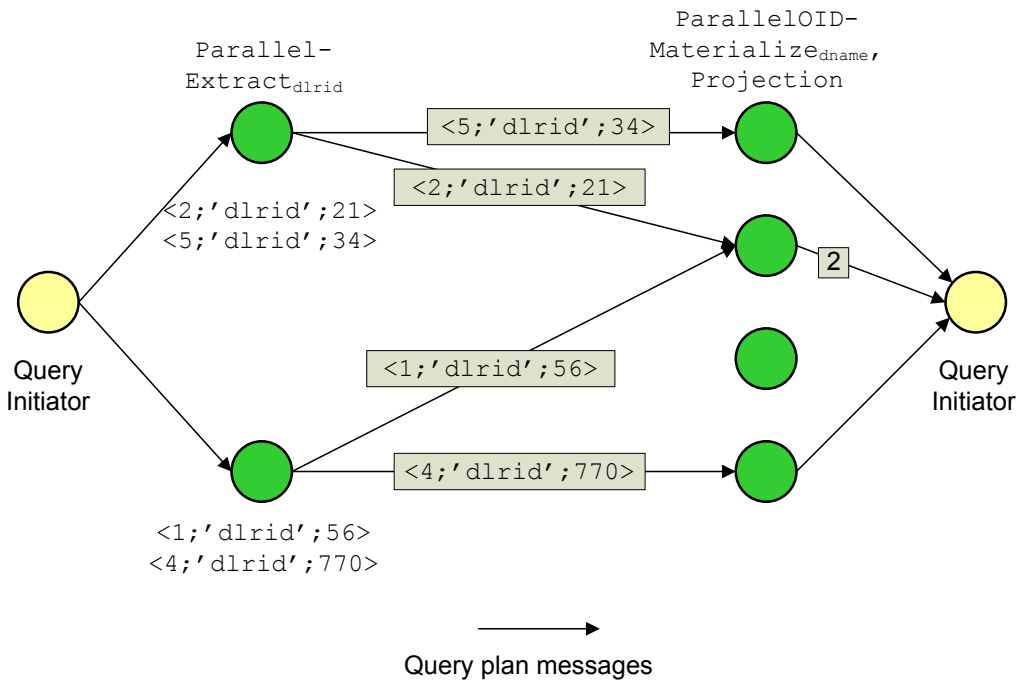

Figure 6.12: Plan cloning operator processing

6.4.3 Parallel Execution of Binary Operators

The serial and parallel execution strategies still evaluate branches of binary operators sequentially (i.e., in postorder). $Join$ is the only such operator in the VQL algebra but others, like union or intersection, could be added. In this section the possibility of executing the left and right branches in parallel will be explored (inter-operator parallelism). This could even be generalized to n-ary operators, but the focus in the following will be on joins.

Depending on the operators used in the branches, they will produce exactly one – if no prefix query or plan cloning operator is used – or multiple query plans each. In both cases the key question is how to synchronize: each plan from the left must be combined with each from the right and the join processed on this data. Consider the logical query plan shown in fig. 6.13. The results of the branches will be available on the peers involved in the processing of $Materialization_c$ and $Materialization_d$. If there are only serial operators in a branch the result will be available on the last peer involved in the processing. Usually it is not known which peer this will be. Fig. 6.14 shows the query plan with parallel operator implementations. The results of the two branches will be available on the peers storing the AV index for c and d in this case. A natural solution would be to send them from the left to the right side with another prefix query on $h(d)$. The joins would then be computed on these peers in parallel. Because of P-Grid's replication, this approach does not work – multiple peers exist for a given path and it can not be guaranteed that the prefix query reaches the same set of peers which processed $\texttt{ParallelAVMaterialize}_\texttt{d}$. A special prefix query could be started to find out one representative set of IP addresses for these peers before the query is started. They could be stored in the plan and used during processing by both branches. A drawback is that the lookup will delay the start of plan processing, and the approach will most likely not be very robust.

Instead, the mechanism for synchronization on a single peer is used (sec. 6.4.1). The principle is the same but the algorithm for merging the data differs and will be described below. While a central component is not ideal for P2P environments, it is deemed sufficient for evaluating the difference parallel join processing makes. Furthermore, this works for all kinds of physical operators and logical query plans, not only prefix query operators.

In practice parallel join processing is triggered as follows: an M^{2}QP containing logical and/or physical operators is generated and cloned. In place of the usual `LocalJoin`

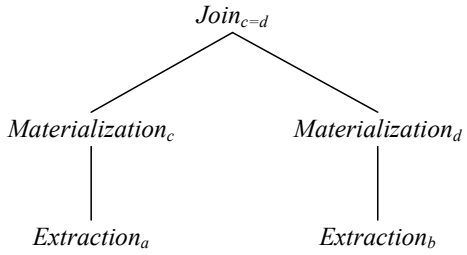

Figure 6.13: A query plan which can profit from parallel join branch execution

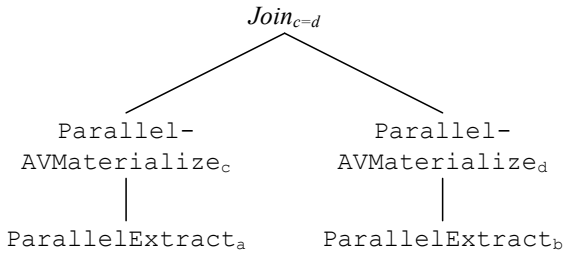

Figure 6.14: The physical query plan using intra-parallel operators

operator `ParallelJoin` is used which extends it by synchronization. In one copy of the plan, all operators from the left branch are marked inactive, in the other copy the right branch. The execution engine will still process plans in postorder but skip the inactive operators which leads to the exclusive execution of the branches. Both plans are then tagged with the same query ID and passed to the engine. When an M^{2}QP reaches the synchronization peer the data it contains (either in the left or right child operator of the join) is appended to the locally stored data from previous plans. Then, the other join side of the current plan is filled in with local data and processing continues. This incremental approach ensures complete join processing even if the branches contain parallel operators generating multiple plans. The local data is discarded after a timeout without any accesses.

6.5 Query Status and Completion

One drawback of the serial and parallel M^{2}QP query processing strategies is the lack of feedback to the user. The result of a serial query may not be available immediately.

It is not known how long processing will take – the user simply has to wait. Messages may get lost, so there is no way to distinguish between long-running and failed queries. For parallel processing, the number of result plans that will arrive is not known. While the result is continuously updated via online processing, the user does not know how *complete* it is. Approaches to remedy these problems will be presented in this section. Implementation is left for future work.

6.5.1 Serial Strategy

Currently, the query processor waits until the result arrives or an error message indicates failure[8]. This message or the query plan itself can get lost and the user will not receive any feedback at all. The key questions in this context are "Is the query still running?" and, more specifically, "How much of it has been completed?", which allows to estimate when the result will arrive.

The simplest solution for the first problem is to consider all queries failed which do not deliver the result in a given time. But successful queries can also take long using the serial strategy, so this timeout would have to be high and the user would still be left without any information during this period. As an alternative, the peers processing the query could send update messages to the initiator ("heartbeats"), thus indicating that the plan is still "alive". While the initiator also has to wait a certain amount of time between each such message during which it does not know anything about the state of the query, this interval is much smaller than the one between the start of the query and the final result. The loss of heartbeats is no big problem, as the next will be sent soon after. Information about the state of plan execution can be included which is very useful, as some aspects of the plan might only be known during query time[9]. Because they can also be used for the parallel execution strategy, heartbeats were implemented in two different forms:

Peer Heartbeat Each time an M^{2}QP reaches a peer, a heartbeat message is sent before processing of the plan begins. Should a peer be contacted multiple times, it will also send multiple notifications. This provides essential "query still alive" information. No message will be sent during local processing, but these periods should not be very long.

[8] only fatal errors cause this – many errors are tolerated ("best effort")

[9] for example, the number of jtuples about to be processed by an operator, which can help to estimate execution time

Operator Heartbeat This type of heartbeat is sent for each operator in the plan when it is first processed. Operators contain a GUID[10] for identification which is included in the heartbeat message, so the initiator can determine how much of the plan has been processed. Note that multiple notifications will be sent when multiple copies of the plan exist in the network, for example after processing of a plan-splitting operator like `ParallelExtract`.

Both types can be enabled on a query basis. They can be used together and provide fine-grained, lightweight feedback. By using its knowledge about the query plan and the physical operators in conjunction with the feedback (including query-time statistics) the initiator can construct a model to provide good estimations for the two questions posed in the beginning ("query still alive" and "percentage completed"). Based on them the user can make an informed decision whether to wait longer for the result.

6.5.2 Parallel Strategy

The challenge for a query processed by parallel operators is to determine when all of the concurrently generated result plans have arrived at the initiator. Synchronization also had to deal with this problem (sec. 6.4.1) and online processing was introduced as a solution, which continuously merges incoming result plans and signals the refined result to the user. A timeout was used to end the query. In a P2P setting it is not possible to exactly calculate the number of plans that should arrive due to the lack of global knowledge and possible failures during transmission. Therefore, approaches for estimation will be presented in the following, which still provide a substantial improvement over a fixed timeout implementation.

Initiator Feedback

The first idea uses additional messages, including heartbeats, sent from the peers during processing to the initiator, which builds a model to estimate how many result plans will arrive.

A query starts with a single plan and each parallel operator duplicates it by a certain factor. `ParallelRanking` will merge these plans again and may generate multiple revi-

[10] globally unique identifier

sions. Therefore, the two key questions are (1) “What is the final revision?” and (2) “How many result plans are part of this revision?”

(1) Revisions are only used for `ParallelRanking` and `ParallelJoin`, which needs not be considered here (see below). Each time a new plan arrives on the synchronization peer it is merged with existing data, processed and the revision number incremented. The final revision contains all plans produced by the operator preceding `ParallelRanking`, so the revision number will be equal to this number. Thus, question (1) becomes equivalent to (2). To determine the final revision number, the number of plans generated by the preceding operator must be estimated[11].

(2) Plan duplication can happen in two ways. First, a plan can be sent to multiple peers with a prefix query. The sender does not know how many peers will receive the plan because P-Grid’s routing layer does not provide this information, but peer heartbeat messages from the recipients could be counted on the initiator to estimate this number[12]. Second, plan cloning can be used to generate a set of plans on one peer and route them, so the sender knows the number and can send it to the initiator. The third approach for parallelism, parallel processing of join branches, does not need to be considered here. While it duplicates the query plan on the initiator, this is offset when the copies are literally “re-joined” by `ParallelJoin` later on.

Fig. 6.15 shows an example query plan using four prefix query operators[13] and fig. 6.16 depicts the execution without any errors, including heartbeat messages[14]. Multiple messages between peers are indicated by a number. The `ParallelExtract` AV index is stored on 2, the first `ParallelAVMaterialize` index on 2 and the second on 3 peers, and a total of 20 peer heartbeat messages are sent to the initiator[15]. They contain the peer’s path and the GUID of the active operator, so for each prefix query operator the number of involved peers can be determined by counting the distinct peers as identified

[11] this operator itself does not have to be a parallel operator – in this context, it simply means that it generates multiple plans for the same query, which can also be caused by a preceding parallel operator lower in the plan

[12] it is an estimation because heartbeats might get lost and peers can fail after sending them without forwarding the query plan

[13] `ParallelAVMaterialize` can only fetch one attribute, so two instances must be used

[14] as in fig. 6.7, operators are associated with the peers where they do their relevant processing, but they also run on other peers – for example, `ParallelExtract` also runs on the query initiator

[15] note that `ParallelRanking` generates 2 revisions, which are completely evaluated and also generate heartbeat messages each, hence the high number

by the path. This eliminates the duplicate messages generated by the processing of the different revisions.

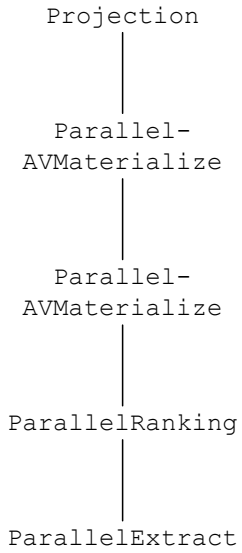

Figure 6.15: Example prefix query plan

The query initiator can now answer the two questions posed using this information and the structure of the query plan. Only one parallel operator is processed before `ParallelRanking` and it is known that it will produce two concurrent query plans, because two heartbeat messages arrived from the peers responsible. Both plans are forwarded to the ranking peer, which produces a new revision for each one, with the second containing the data from both plans. After that, two parallel operators again cause duplication. Both of the first two `ParallelAVMaterialize` peers send each plan revision to all three peers of the second `ParallelAVMaterialize` for 6 total plans per revision. The initiator therefore knows that 12 plans will arrive and those with revision 2 are final.

For plan cloning operators a different approach is used. Fig. 6.17 shows a plan with a `ParallelOIDMaterialize` cloning operator. First, the plan is routed to the peers storing the AV index for the extraction attribute. This generates 2 peer heartbeat messages. `ParallelOIDMaterialize` is executed next. It generates 3 and 2 messages on the peers and sends a feedback message to the initiator with this information, which adds them up to determine the number of concurrent plans[16]. The initiator can determine

[16] operators which split the plan recursively multiple times (currently only `ParallelQgramExtract`) can also be handled, but are not considered here

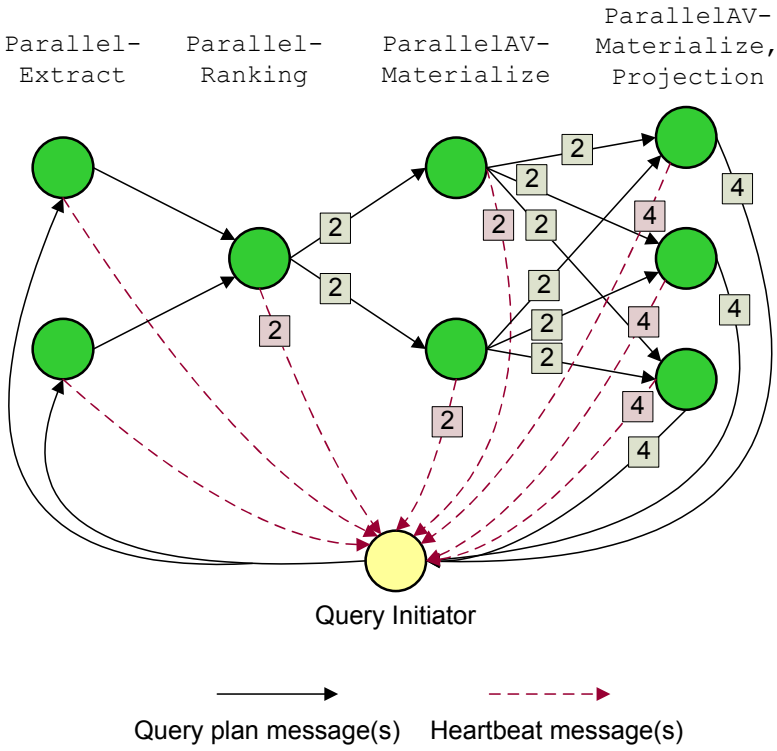

Figure 6.16: Messages and heartbeats sent during parallel query execution

that it must wait for two feedback messages by examing the received information (peer heartbeats or plan cloning feedback) from the preceding operators in the plan (only `ParallelExtract` in this case).

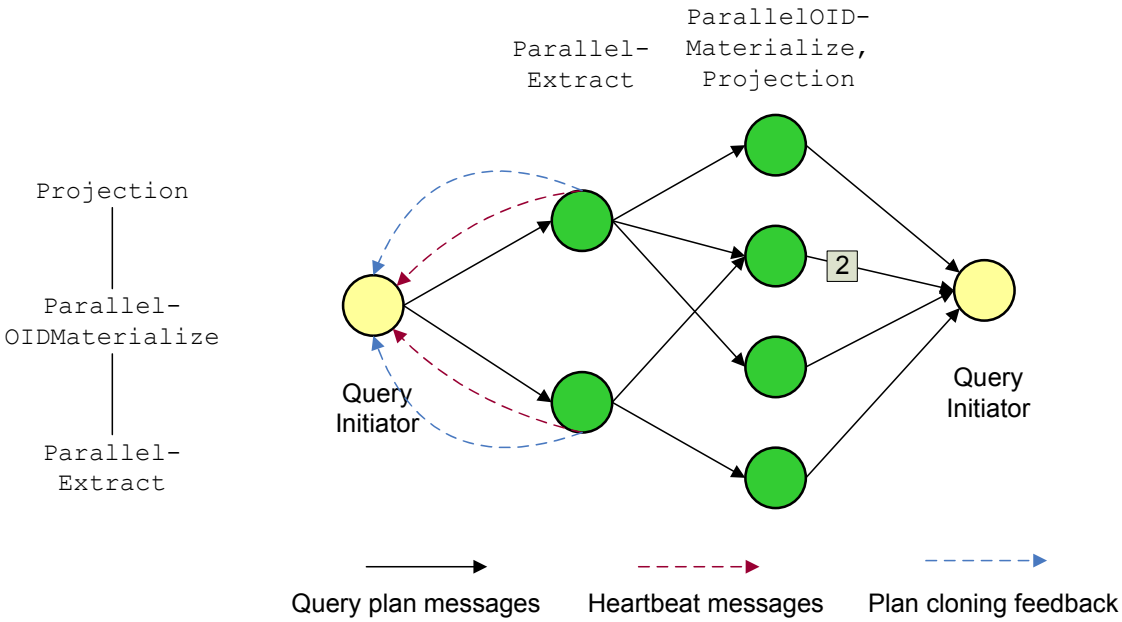

Figure 6.17: Example plan cloning query plan

Not all peer heartbeats are depicted in fig. 6.17. The peers storing the OID index also send them, and they can also be used in this case for completeness estimation. In practice it would be best to switch them off for plan cloning operators and use the above approach as it generates less messages.

Fig. 6.18 shows the algorithm for estimation of the final revision and number of result plans (`estimateStatus()`). It assumes that each operator stores peer heartbeat or plan cloning feedback information. It simply uses the available information and does not wait until all feedback messages have arrived, but this can also be implemented using the central `maxMessages()` method on subplans in the way described above.

Waiting with a timeout is still required. Consider the prefix query example plan in fig. 6.16. When a plan is received on one of the final `ParallelAVMaterialize` peers it will send out the heartbeat, process the plan and send the result to the initiator. Therefore, heartbeat and result will arrive at about the same time – the result may even arrive first. The problem is that heartbeats are sent relatively late – only when the operator

```
void estimateStatus(Plan p) {
    (subplanAboveRanking, subplanBelowRanking) = splitPlanOnRanking(p)
    revision = maxMessages(subplanBelowRanking)
    plans = maxMessages(subplanAboveRanking)
    ...
}

int maxMessages(Plan p) {
    i = 1
    Operator[] opQueue = p.postOrder()
    for (op in opQueue) {
        if (op.type == PLAN_CLONING) {
            // calculate number of plans at this point from
            // received feedback messages
            msgs = op.getPlanCloningFeedbackMsgs()
            i = 0
            for (m in msgs) {
                i += m.getNumClonedPlans()
            }
        } else {
            // a prefix query simply multiplies the number of plans by the
            // number of peers storing the index - each sends a heartbeat
            i = i * op.getNumPeerHeartbeats()
        }
    }
    return i
}
```

Figure 6.18: Estimating the number and final revision of result plans

is already running – and to make sure to receive all of them, a timeout must be used again. `ParallelRanking` mitigates this problem because earlier revisions will be processed and the corresponding heartbeats sent, allowing the initiator to acquire knowledge about the number of peers at each level . When the final revision is processed, it is very likely that all heartbeats from the first revision have arrived. The initiator already has a good estimate for the number of plans based on the early revision and does not have to wait for the heartbeats of the final revision. For plan cloning operators only the feedback messages of the final revision provides accurate information, but the number of generated plans can be sent to the initiator before sending them to the peers for processing, so it will arrive earlier than the results and the timeout can be lower.

Prefix Probing

The previous approach showed that acquiring knowledge early is important so timeouts do not have to be used to wait for feedback. Prefix probing adheres to this by acquiring information about prefix operator peers at the beginning of the query.

The technique is illustrated in fig. 6.19. The initiator first sends out a new type of prefix query message for every prefix query operator, addressed to the prefix it operates on. Each peer receiving it answers to the initiator, which can count the number of peers, similar to the peer heartbeat approach above. At the same time the query is started. Because the probing is done in parallel and at the beginning, not during query execution, the number of peers each operator is processed on will be known early and there is no delay for operators at the top of the plan. The number of expected plans can be calculated from this and should be relatively stable when the first result plans arrive, so a progress indicator could be displayed. After the estimated number of plans have arrived it is safe to end query execution without further waiting. This approach can not be used for plan cloning operators as is, because the number of plans generated will only be known during processing, but the aforementioned solution of sending feedback messages during processing to the initiator can be applied. When the operator occurs at an early point in the operator queue (i.e., at the bottom of the plan), the information will arrive in time and no long timeout is necessary. The methods in fig. 6.18 are also used for estimations.

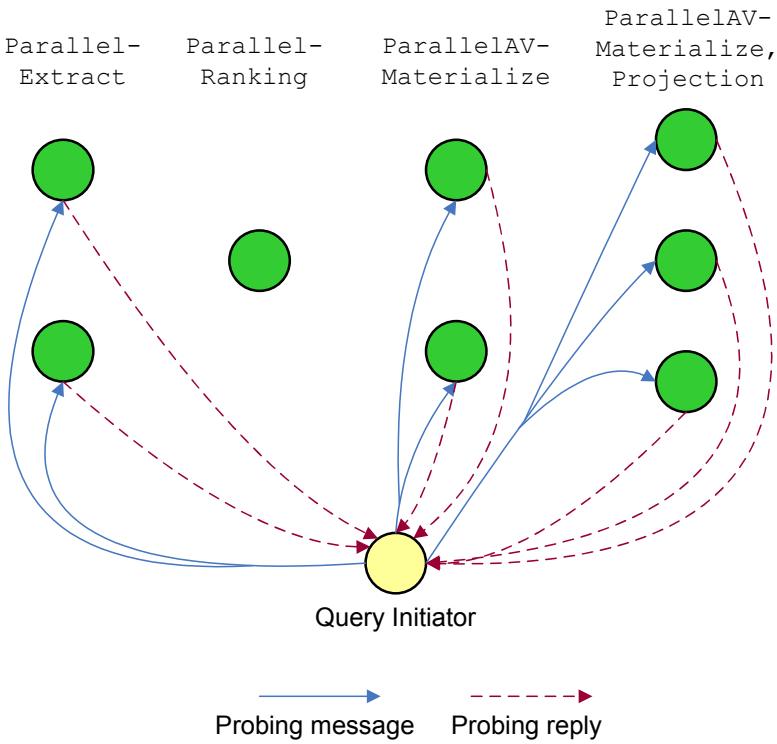

Figure 6.19: Determining the number of peers with prefix probing

Plan Tagging

In this third approach, no additional messages are sent. All information about the execution of the query is stored in the plan. Each prefix operator in each plan is tagged with the path of the peer that processed it. The initiator extracts the information from the incoming result plans and determines the number of distinct peers for each operator. If there are no plan cloning operators in the plan, the product of the peer counts equals the number of results. Plan cloning operators can also be handled. Instead of sending a feedback message as above, the number of generated plans is simply stored in the plan associated with the operator. In contrast to the heartbeat approach, there is no feedback until the first plan arrives. When the plan contains a blocking operator but no plan cloning operators this is no problem, as the information obtained from earlier revisions can be used as described in sec. 6.5.2. The peer numbers will be known when processing of the final revision is underway. For other plans one of the other approaches is better suited.

Fig. 6.20 illustrates tagging for prefix query operators. Letters have been used to represent the distinct paths of the peers. The AV index for `ParallelExtract` is stored on two peers A and B, for `ParallelAVMaterialize` on C, D and E. Therefore, each plan will pass through A or B and after that either through C, D or E, yielding 6 possible combinations of tags, one attached to each of the result plans sent to the initiator. When plans B-D and B-E arrive, it is known that there is at least one extraction and two materialization peers. If A-D arrives next, it is clear that one more peer exists at the extraction level and another plan A-E must also exist. This way, a lower bound can be determined. Fig. 6.18 can be used for estimation once the relevant information has been extracted from the result plans (distinct paths correspond to the peer heartbeats).

Discussion

All three methods can be implemented with relatively little overhead. Few information needs to be transmitted, so the increase in bandwidth is negligible. The additional messages received in the first two approaches might place some load on the initiator, so tagging can be a better choice. Tagging and initiator feedback should perform reasonably when a blocking operator is present. Prefix probing is expected to perform best. For plans with blocking operators it also generates less traffic than initiator feedback, because it only queries prefix query operators once, while heartbeats are sent for every revision. Plan cloning operators are the bottleneck for all three approaches. As the number of generated plans is only known at processing time, the generated estimations

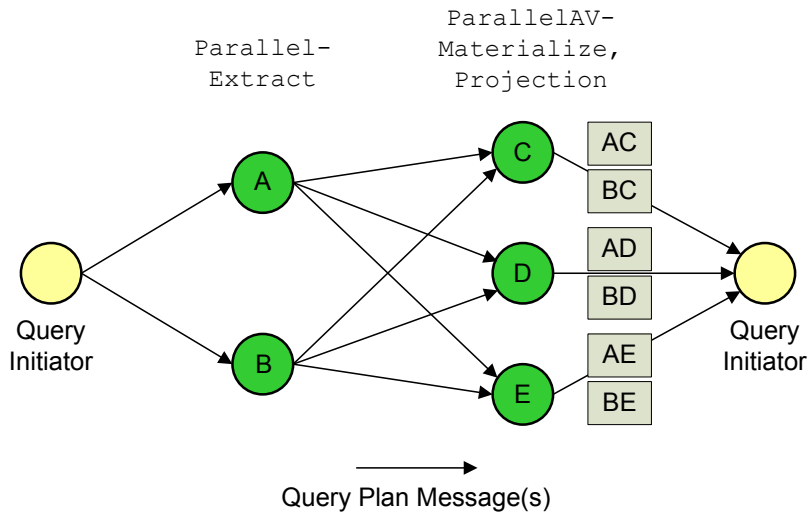

Figure 6.20: Tagging plans to estimate the number of result plans

are often “late”, because feedback messages and results arrive at about the same time. A test implementation could provide good insights into the general applicability of these approaches in a real-world system.

6.6 Summary and Outlook

Serial and parallel plan execution strategies based on the idea of Mutant Query Plans have been presented in this chapter. Initial tests show that they perform well in a P2P setting. In the future, plan execution should become even more flexible. The postorder approach has already been extended by parallel processing of binary operators. In addition, each peer should be able to decide on its own where to route the plan next, for example by consulting local statistics. It should also be possible to rewrite the plan locally. Three possible approaches for this are presented in [PM02a]: *consolidation*, *absorption* and *deferment*. Using properties of the logical operators like associativity and commutativity, they restructure the plan for earlier or later processing of operators, thus reducing the size during transmission. For example, a cartesic join will increase the size and it is usually better to just store the data of both sides and delay processing as long as possible. To increase parallelism and lower latency in cases where local processing takes long a plan can be fragmented: instead of waiting for the completion of

a given operator on all tuples, a plan fragment is forwarded for each n tuples processed. More details on this *pipelined parallelism*-approach can also be found in [PM02b].

7 Operators

This chapter deals with the physical operators available in CouPé. The first section reviews their general properties. Next, each operator is examined in detail. One section focuses on the local operators which do not need to be routed, a second on DHT operators which are further categorized into serial and parallel operators. With all operators known the available query planners for mapping logical to physical operators are documented. Possible extensions to the available implementations, including aggregation, are discussed last.

7.1 Overview

A physical operator as available in CouPé is *executable code* and stores jtuples processed by itself; interfaces are provided for the parent operator to access them. In addition to data processing, operators perform routing and plan modification tasks. An operator might have to visit a number of peers, and it must control this process by specifying the destination peers. A state machine keeps track of this internally. All internal data structures are transmitted as part of the serialized query plan. Note that the code itself need not be included, only the types of the objects used need to be stored because the code itself is available on all peers.

Jtuples (sec. 4.2.2) have been used to preserve the structural relationships between triples established by the operators. Equivalent data structures in the operators can store them. Furthermore, each operator must know which component of a jtuple to process. This is specified by the VQL notation (sec. 4.2.3) and can generally be any OID, attribute or value. Query planners parse this information and pass it to the operators when instantiating them (sec. 5.3). The implementation supports string, integer and float data. Only the first two are used in this work. If the data types do not match, the offending jtuples are silently discarded.

7.2 Local Operators

The operators presented in this section only operate on the jtuples supplied by their child operators. They do not cause routing of the plan and do not access the local storage.

LocalSelection This implementation of the logical $Selection$ operator supports all operations: $<$, $>$, $=$, $\leq$, $\geq$, $\neq$ and $\sim$ (similarity). In addition to this the argument for comparison, in the case of a similarity selection the maximum distance, and the component to process must be specified. For each jtuple provided by the child operator the component is extracted and the operation evaluated. The selected jtuples are stored in the operator.

LocalJoin This operator evaluates joins with a simple nested loop. For both sides the jtuple components to compare must be specified. This is usually a value or attribute component (instance/schema level operation). The two sides can operate on different levels. For each jtuple combination the specified comparison operation is evaluated. The same operations as for `LocalSelection` are supported, including similarity joins. A cartesic product can also be generated. Lastly, VQL allows the specification of multiple conjunctive predicates for one $Join$ and this operator provides a short-circuiting AND implementation for this. Jtuples from both side are combined to a new jtuple while retaining all of their data as required (sec. 4.2.2).

LocalRanking The `LocalRanking` operator is the most complex of all, as it has to implement the three different ranking functions $Minimum$, $Maximum$, $Nearest\ Neighbor$ and additional `LIMIT`/`OFFSET` clauses. For all three functions the component of the jtuple on which to operate must be specified. $Nearest\ Neighbor$ accepts two additional parameters: the reference object (a string or integer) and optionally a maximum distance l. For each candidate jtuple the difference d to the reference object is computed. For numbers this is the euclidian distance, for strings the Levenshtein edit distance. If $d > l$, the jtuple is discarded. All remaining jtuples are sorted in ascending order on the distance. $Minimum$ and $Maximum$ sort all the jtuples of the child operator by the component specified.

When multiple functions are specified they are evaluated in a nested fashion: the first function is evaluated as usual. The next function is computed on isolated blocks of data

where the component on which the first function sorted is equal. Therefore, it is ensured that the sorting order of this component is not changed by the succeeding function. The next function will keep the components sorted by the previous two functions intact, and so on.

After sorting the `LIMIT` and `OFFSET` clauses are evaluated. `OFFSET` skips the specified number of jtuples starting from the beginning, `LIMIT` cuts off all jtuples beyond a certain number. Lastly, each jtuple in this sorted list is numbered. This preserves the order in cases where the plan is split by subsequent parallel operators and must be merged again on the initiator, which can determine the correct order by the numbers. As only one $Ranking$ will appear in VQL plans, this solution is sufficient.

Projection `Projection` is the operator at the top of the tree. It is different from other operators because it stores the generated data – the result of the query plan – in a special format similar to the tabular structure of SQL results instead of jtuples. A list of components to extract from each jtuple must be specified. A component can be an OID, attribute or value. They are extracted and the result is generated. Each jtuple corresponds to a row and each component to a column.

7.3 DHT Operators

The operators in this section use the DHT for routing, most also retrieve data from it.

7.3.1 Serial Operators

All operators without intra- or inter-operator parallelism are considered serial operators. They route the plan sequentially from peer to peer without any plan duplication.

Extract This operator extracts triples from the DHT using the AV index (sec. 4.1.1). It can perform three operations:

- fetch all triples ($routeKey =''$, an empty key)
- fetch all with attribute att ($routeKey = h(att)$)

- fetch all with attribute att and value val("direct key lookup", $routeKey = h(att) \circ h(val)$)

First, $routeKey$ is calculated and the plan sent there. If the receiving peer's path $localPath \subseteq routeKey$, all requested data resides there. Otherwise, the peer is just a member of a subtree of peers whose paths have $routeKey$ as prefix, and all others must be contacted as well. Therefore, all other possible peers' paths are generated and routed to, but not all of them must exist. On the other hand, each path can be a tree again, so this process must be repeated recursively. The algorithm for this is shown in fig. 7.1. The recursion is implemented with a routing queue.

Fig. 7.2 shows a virtual P-Grid tree and the processing of `Extract` on an attribute which hashes to $routeKey = 1$. The three red peers must be contacted to gather all data. Initially, the routing queue contains only 1. `process()` is invoked and the plan is sent to this key, but no exact-matching peer exists. P-Grid will send the message non-deterministically to one of the red peers in this case. Assuming it will be peer 110, the operator detects that the local path is longer than $routeKey$. It generates the remaining paths by keeping the first bit as specified by the $routeKey$ constant while permutating the remaining two to yield paths 100, 101 and 111 which are scheduled for routing while 1 is removed from the queue. The plan is next routed to 100 and reaches peer 10. In addition to 100, 101 can also be eliminated from the routing queue as 10 is also responsible for this prefix. Processing ends at peer 111.

This way the expected cost for `Extract` is $O(m * logN)$ for N total key space partitions and m key space partitions responsible for the data requested. For a direct key lookup, $m = 1$ when the hash function generates keys which are longer than the longest path in the network (an implicit requirement of P-Grid). The expected number of participating peers can be used to determine the maximum possible path length.

OIDMaterialize This operator utilizes the OID index (sec. 4.1.1), parameters are the attributes to add to each jtuple. It is also possible to fetch all available attributes without naming them, which is required by some VQL queries. For each jtuple supplied by the child operator the plan is routed to $k = h(OID)$. Multiple OIDs can be part of one jtuple (for example, after joins), so the correct one must be identified by the query planner. The "prefix problem" illustrated above also applies here – it can happen that multiple peers are responsible for k when the keys generated by the hash function are not long enough. In this case all other peers are contacted as a workaround, but the better solution is to adjust the hash function[1].

[1] requiring the rebuilding of the grid

```
// the paths that still need to be processed
// are queued up here

// initialized with the hash key of the
// attribute to extract
String[] pathsToRoute = [ h(att) ]

void process() {
    while (pathsToRoute.length > 0) {
        String routeKey = pathsToRoute[0]
        routeTo(routeKey)

        // check the path of the peer we arrived at
        localPath = PGrid.getLocalPath()
        if (localPath.length > routeKey.length) {
            // local peer part of a tree
            // generate all other possible peer paths and schedule for routing
            pathsToRoute.add(getAdjacentPaths(routeKey, localPath))
        }
        // remove all paths which this peer is responsible for,
        // so we do not route there again
        for (path in pathsToRoute) {
            if (path.startsWith(localPath)) {
                pathsToRoute.remove(path)
            }
        }
        // extract triples from av index
        Triple[] t = tripleMgr.getTriples(h(att), "av")
        ...
    }
}

String[] getAdjacentPaths(routeKey, localPath) {
    // determine the common prefix of routeKey and localPath
    // keep it constant in localPath, generate all permutations of
    // the remaining bits and return these paths except for localPath,
    // which has already been contacted
    ...
}
```

Figure 7.1: Routing algorithm for `Extract`

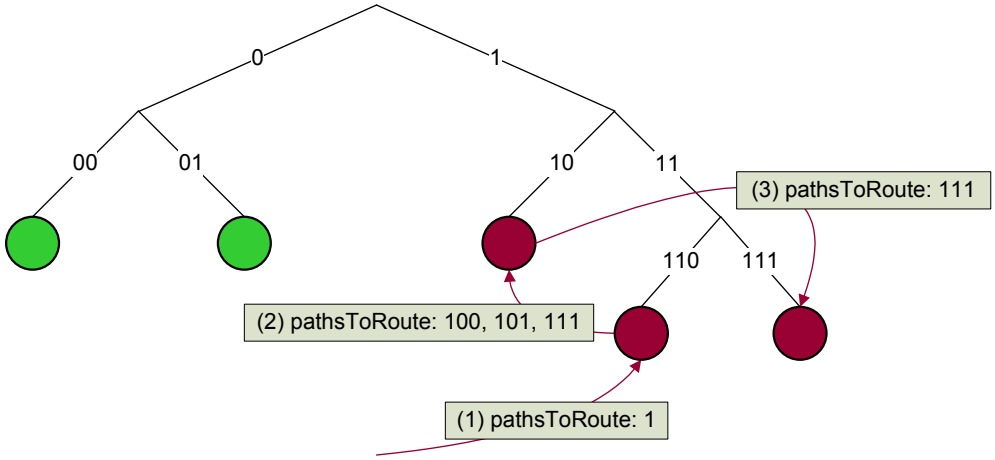

Figure 7.2: Example: virtual P-Grid tree and processing of the `Extract` operator

OptOIDMaterialize One peer may store $Materialization$ data for multiple jtuples. When they are not stored adjacently in the child operator, `OIDMaterialize` will route to the same peer twice or more. For example, if peer A is responsible for jtuples 1 and 3, peer B for jtuples 2 and 4, the plan would be routed to A, B, A and B. `OptMaterialize` removes this inefficiency by contacting a minimal number of hosts only.

QgramExtract This operator implements similarity search on strings using the q-gram indexes described in sec. 4.1.2. The Value similarity index is not used in this work, but it is relatively easy to extend this operator to handle it as well. Parameters are the search string s from which the edit distance to the candidates will be calculated and the maximum distance d (an integer). Furthermore, the attribute whose values should be searched can be specified and the AV similarity index (instance level) is used in this case. Otherwise, a search on the attributes themselves is performed using the Attribute similarity index (schema level).

The basic algorithm for q-gram search has been described in detail in sec. 4.1.2 and will only be summarized in the following. First,the search string s is converted to lower case to perform a case insensitive search (q-grams indexes are also stored in lowercase). $d+1$ non-overlapping q-grams q_i are generated from it as described in sec. 4.1.2. If the string is too short, all (overlapping) q-grams are extracted instead, which might not be able to find all results. Each q-gram is hashed to P-Grid keys, producing $k_i = h(att) \circ h(q_i)$ or $k_i = h(q_i)$ for instance and schema queries, respectively. Finally, duplicate keys

are eliminated, the plan is routed to those remaining and the data stored on the peers is extracted. Similar to `OptOIDMaterialize`, all other keys scheduled for routing are also processed when they are stored on the peer. This is the case when the peer's path is a prefix of the key. The "prefix problem" and its solution (sec. 7.3.1) also apply here. Different q-grams may reference the same triple, so a set is used as internal data structure to prevent duplicates.

Up to now, false positives have not been eliminated. Q-gram results must always be post-processed to eliminate them. However, it would not make sense to implement this filtering functionality again, as it is already available in a more generalized form in the `LocalSelection` operator. The forward processing approach presented in sec. 6.3.1 comes in handy at this point. `QgramExtract` can simply be combined with a filtering `LocalSelection` on top, yielding one virtual operator. Each time `QgramExtract` has fetched data on a peer, `LocalSelection` is immediately processed to drop any false positives. This reduces the size of the plan just before it is sent to the next peer.

Further optimizations are possible. The q-gram generation algorithm should supply the q-grams with the lowest selectivity to reduce the size of intermediate results generated, even if these results are only processed locally. Non-deterministic generation could be used to randomly create different q-gram sets for a search string and thus balance load among the peers storing the index. Also, more than $d + 1$ q-grams could be used for redundancy, combatting the failure of peers.

7.3.2 Parallel Operators

Parallel operators make use of prefix queries (sec. 6.4.1), plan cloning (sec. 6.4.2) and parallel execution of operator branches (sec. 6.4.3) for concurrent processing.

ParallelExtract This is the parallel version of `ExtractOp` with the same functionality. Prefix queries are used to parallelize execution: instead of sequentially routing the plan to all peers responsible, P-Grid's routing layer automatically does this in parallel. For an attribute stored on n peers as many plans are sent.

ParallelOIDMaterialize This is the combination of `OptOIDMaterialize` and plan cloning, the parameters are identical. For every jtuple supplied by the child operator, the appropriate OID is hashed to $k = h(OID)$ and a new plan containing only this jtuple is routed

there. From this point on all these parallel plans are processed like in the serial version: when k addresses a subtree, the plan is routed to all other peers in it. This can not be done in parallel because attributes might be stored on different hosts and splitting the plan at this point would not correctly materialize the jtuple. The operator should be used in cases where many attributes need to be materialized at a time (which will very likely be stored on the same peer) and the number of jtuples is low.

ParallelAVMaterialize This variant uses prefix queries for parallelism. Unfortunately it is not possible to pose a meaningful range query on the OID index. Instead, the AV index is used in the same way as for `ParallelExtract`: the plan is routed in parallel to all peers of the required attribute, $k = h(att)$. At each peer, the child operator in the plan contains all data to be materialized. But only those with local $Materialization$ partners can be processed. All others are removed from the plan, but as the AV index contains all triples for the requested attribute, it is ensured that they will be processed on one of the other peers. The drawback is the overhead incured by shipping all jtuples in the child operator to all peers. Furthermore, only one attribute can be materialized at a time this way, in contrast to operators using the OID index. $Materialization$ of multiple attributes is handled by multiple `ParallelAVMaterialize` in sequence, but the case where all available attributes must be added without naming them cannot be handled.

This operator is ideal if only a few attributes need to be materialized, or if the number of tuples is high and each one is relatively small. In this case the bandwidth overhead is no big drawback, but the speedup is substantial, because there is no need to route to many peers in sequence (`OptOIDMaterialize`) or to create a plan for each jtuple with substantial overhead (because the jtuples are small – `ParallelOIDMaterialize`).

ParallelRanking When parallel operators are used, synchronization must happen before $Ranking$ can be evaluated (sec. 6.4.1). While the execution engine handles the merging of plans, it is the responsibility of the operator to route to a synchronization peer first. As pointed out in sec. 6.4.1, this peer cannot be specified by a P-Grid path because of replication. Therefore, a concrete peer is addressed – which can be different for every query – and its IP address and port are additional parameters of this operator. In all other aspects `ParallelRanking` is identical to `LocalRanking` (sec. 7.2).

ParallelJoin This operator extends `LocalJoin` with synchronization before processing happens. The coordination peer is specified by IP address and port. The plan is routed

there and the execution engine handles merging with other parallel plans. This process is explained in detail in sec. 6.4.3.

Note: Both `ParallelRanking` and `ParallelJoin` extend existing operators with synchronization. This functionality could be outsourced to a separate operator which is then inserted before `LocalJoin` or `LocalRanking` operators. Due to time constraints this was not implemented.

ParallelQgramExtract This parallel variant is similar to `QgramExtract` and is also used in conjunction with a filtering `LocalSelection` on top (sec. 7.3.1). Instead of routing to the q-gram hashes in sequence, one plan is generated and routed in parallel for each. Due to the "prefix problem" further routing might be required. It is handled similar to the solution described in sec. 7.3.1 – all other peers are also routed to – except that the messages are sent in parallel again. This process is repeated recursively if necessary, leading to a tree-like dissemination of messages. Parallel q-gram processing leads to duplicates. For the search for "mistake" with $d = 1$ in fig. 4.4, two q-grams "$mi" and "sta" are generated. Triple 1 is associated with both of these q-grams. The operator generates two messages and both will fetch the triple. This can be solved by filtering the final result at the query initiator for duplicates.

ParallelQgramJoin Consider the plan depicted in fig. 7.3 where a similarity join needs to be processed. One subtree contains just an $Extraction$. Without loss of generality it is assumed to be the right subtree, as the child operators can be swapped. The data processed by the $Join$ (a and b) is fetched in the left and right branches, respectively. It can be data on instance (values) or schema level (attributes). The left side will be evaluted before the right side in the basic M^{2}QP approach. Therefore, instead of fetching all data referenced by b and processing the $Join$ on it, a *ship where needed*-approach becomes possible, which uses the data available in the left branch together with information from the $Join$ predicate to directly ship tuples to peers where results can be expected.

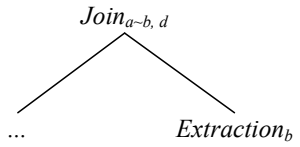

Figure 7.3: A logical plan containing a similarity join

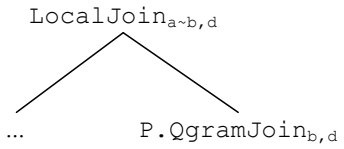

Figure 7.4: The plan before the start of execution

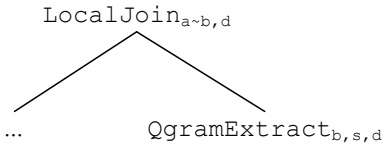

Figure 7.5: A plan generated by `ParallelQgramJoin`

`ParallelQgramJoin` implements this by instantiating the right-side *Extraction*. It is initialized with the data it should fetch (b) and the similarity distance d of the *Join* operation. All other operators are also instantiated. The resulting generic query plan is shown in fig. 7.4. Processing starts and the left side is evaluated before control is passed to `ParallelQgramJoin`, which requests the jtuples t_i stored by the left child operator of `LocalJoin`. For each one the query plan is cloned, modified and returned to the execution engine for routing. One of these new plans is depicted in fig. 7.5. `ParallelQgramJoin` is replaced by `QgramExtract` with the parameters b, s and d. s is the search string extracted from the jtuple t_i (an attribute or value, i.e., the component addressed by a). At this point, `ParallelQgramJoin` is finished and in all the cloned plans execution continues with `QgramExtract`, which routes the plan to all peers where it can fetch matching data for the provided s. In contrast to the usual application of the q-gram operators it is not necessary to use an additional `LocalSelection` operator, as `LocalJoin` will filter out any mismatching tuples anyway.

This operator is particularly efficient when the left side contains relatively few tuples, or the data referenced by the right-side *Extraction* is extensive. Instead, few tuples are shipped to the matching data, saving bandwidth. It can also be used for equijoins as a side effect, because q-grams can also be used for exact matching. While only string data is handled, it can easily be extended to integers (sec. 7.6.2). Furthermore, this technique can be applied to more complex query plans with operators on top of the right-side *Extraction*, but preceding *Join*, for example a *Selection*.

Schema similarity joins are the most important application for the operator. All triples in the network would have to be fetched on both sides of the join, which is impossible

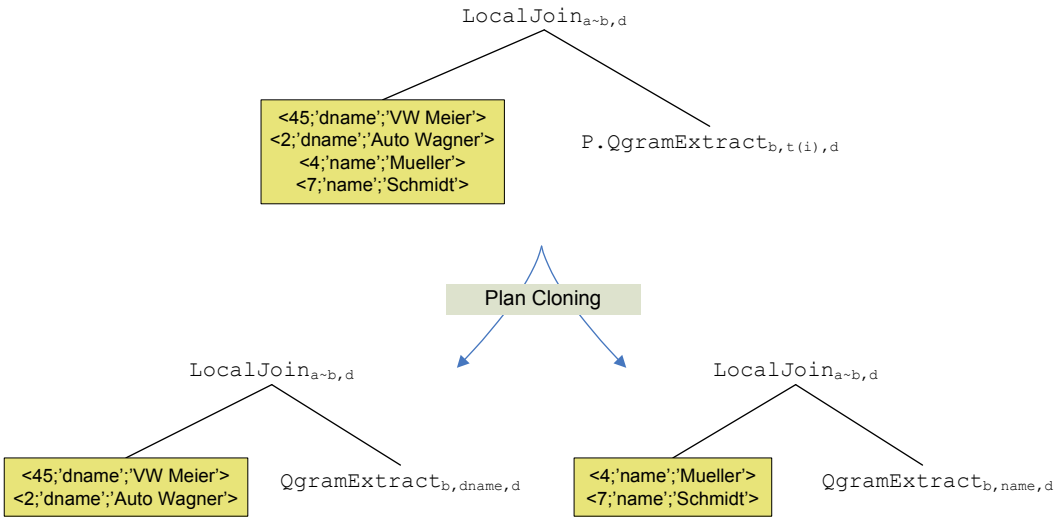

Figure 7.6: Similarity join on schema level using `ParallelQgramJoin`

in most cases. `ParallelQgramJoin` provides a substantial improvement over this. As a further optimization, identical attributes or values can be grouped together and sent in one plan. This reduces the number of messages by a large amount. Fig. 7.6 shows an example for a similarity schema join. The left side of the plan (referenced by a in the join) contains all triples with an attribute similar to `name` which should be joined against all other triples in the network on attribute level with distance d. To efficiently process this query, the data on the left is grouped by attributes and two distinct plans with the two distinct search strings `dname` and `name` are created. `QgramExtract` will operate on the Attribute similarity index to find matches.

ParallelQgramJoin2 This is the same as above, but for further parallelism, `ParallelQgram` is substituted instead of the serial version. This can cause duplicates (sec. 7.3.2).

7.4 Query Planner Operator Mappings

Table 7.1 gives an overview of all available query planners and how they map logical to physical operators. A parent-child operator relationship is denoted by “Parent + Child”.

In the case of forward processing, "$\circ$" is used instead. For layout reasons, "Parallel" is abbreviated by "P.", "Local" by "L." in some cases.

The basic planner has already been mentioned. It instantiates all operators with serial variants and is usually applied last to give other planners a chance to apply their mappings and also to ensure that all operators are resolved. There are different physical mappings for $Extraction$, depending on the scope (instance or schema level). When an equi $Selection$ on the extracted data follows, a "direct key lookup" operation can also be used which will usually find the results in $O(logN)$ time (sec. 7.3.1). The q-gram planners must handle different logical operators for similarity operations on instance on schema level. In the latter case, $Extraction$ contains the predicate, in the former an additional $Selection$ must be parsed. The parallel planner combines three others for convenience and backward compatibility. `ParallelQgramJoinOp` and `ParallelQgramJoinOp2` can be considered hybrid operators because they implement aspects of $Extraction$ and $Join$. Nevertheless, their main task is data localization, and in the plan they are also substituted for $Extraction$, so they have been associated with this operator. No planner is provided for `ParallelJoin`. It did not fit the existing framework as the plan must be cloned on the initiator, but query plans using it can easily be constructed with few lines of code, which is sufficient for this work.

All in all, these planners can be used for a very modular and flexible query planning process. For a given plan, different ones can be applied and tested with ease. Serial and parallel operators can be freely combined. These features make a thorough evaluation of the execution engine and the operator implementations possible, but also provide enough leeway for future extensions.

7.5 Building Custom Query Plans

When functionality which is not yet expressable in VQL (for example, a new operator) should be tested or when special techniques like `ParallelJoin` need to be used it is often easier to construct a plan by assembling operators with a few lines of code than to implement a custom planner. All required operators can simply be instantiated with the required parameters and then linked together. The root operator, which references the plan, is passed to the execution engine for processing. It is also possible to pre-load data in `Extract` which will be processed by the parent operator immediately. This is used in one test in chapter 9.

Query Planner	Logical Operator	Physical Operator
Basic	*Extraction*	`Extract`
		`LocalSelection` ∘ `Extract`[1]
	Extraction + *Selection*	`Extract`[2]
	Materialization	`OIDMaterialize`
	Selection	`LocalSelection`
	Join	`LocalJoin`
	Ranking	`LocalRanking`
	Projection	`Projection`
ParallelExtract	*Extraction*	`ParallelExtract`
		`LocalSelection + P.Extract`[1]
	Extraction + *Selection*	`ParallelExtract`[2]
OptOIDMaterialize	*Materialization*	`OptOIDMaterialize`
P.AVMaterialize[3]	*Materialization*	`ParallelAVMaterialize`[4]
P.OIDMaterialize	*Materialization*	`ParallelOIDMaterialize`
ParallelRanking	*Ranking*	`ParallelRanking`
Qgram	*Extraction*[5]	`L.Selection` ∘ `QgramExtract`
	Extraction + *Selection*[6]	`L.Selection` ∘ `QgramExtract`
ParallelQgram	*Extraction*[5]	`L.Selection` ∘ `P.QgramExtract`
	Extraction + *Selection*[6]	`L.Selection` ∘ `P.QgramExtract`
ParallelQgramJoin	*Extraction*	`ParallelQgramJoin`
ParallelQgramJoin2	*Extraction*	`ParallelQgramJoin2`
Parallel	combines the planners for ParallelExtract, ParallelAVMaterialize and ParallelRanking	
-	*Join*	`ParallelJoin`

Table 7.1: Implemented query planners and their mappings

[1] for similarity *Extraction* on the attribute

[2] "direct lookup" – for *Extraction* of a concrete attribute followed by an equi selection on the values

[3] can only be used if the attributes to be materialized are known by name, see sec. 7.3.2

[4] one physical operator for each attribute

[5] for a similarity *Extraction* on the attribute

[6] for an *Extraction* of a concrete attribute followed by a similarity or equi string *Selection* on the values

7.6 Future Work

This section discusses ideas for new operators and suggests improvements for existing ones which have not been implemented yet.

7.6.1 Grouping/Aggregation

One important feature currently missing is grouping in conjunction with aggregate functions. VQL does not yet support this, but it should be easy to implement at the language level, for example like this:

```
SELECT x, AVG(y) where { <o;'name';x> <o;'len';y> } GROUP BY x;
```

x is the grouping variable, AVG one example of an aggregation function f and y its input. Only operations on the instance level will be considered, but the aggregation function could be easily extended to process attribute and OID data as well. Minor modifications must be made to the way jtuple data is stored in the operators, so the semantics of groups can be represented efficiently.

For the serial strategy, it is only necessary to fetch the involved triple data before processing grouping and aggregation. Fig. 7.7 shows a serial example query plan. For the parallel strategy, three different solutions will be discussed next.

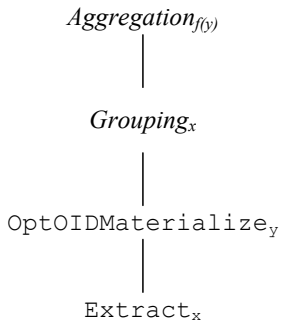

Figure 7.7: Aggregation using the serial strategy

To correctly process aggregations, all data for a group must be available on one peer, which can be achieved by explicit synchronization or by using an index that already stores the data clustered correctly. The AV index is the logical choice for the latter. The query plan in fig. 7.8 uses this approach. `ParallelExtract` fetches x, and it is ensured that triples with the same values are on the same peer because they have identical hash keys – the data is already grouped. The grouping operator must only rearrange the data to fit the internal representation. Next, the second attribute must be materialized. Using `ParallelOIDMaterialize` would destroy the groups, therefore, `ParallelGroupMaterialize` is introduced. For each group it generates a message containing the corresponding jtuples and materializes y on each sequentially, therefore keeping the groups intact. After that, aggregation is performed. This results in as many messages as there are groups.

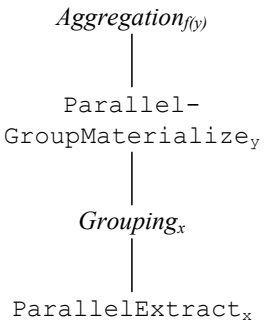

Figure 7.8: Aggregation using the AV index

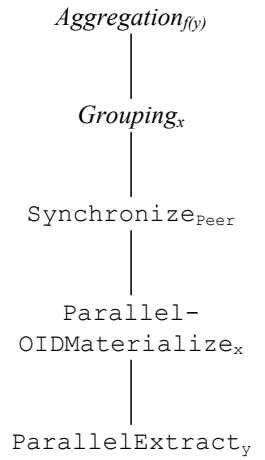

Figure 7.9: Aggregation using synchronization

This approach might not be feasible when there are large groups, because sequentially materializing the jtuples will take long. The alternative is to use `ParallelOIDMaterialize` and synchronize later. This is shown in fig. 7.9. The `Synchronize` operator simply routes the plan to a predetermined peer where it is merged with previously received plans and processed further. An incrementing revision number is used to distinguish results. This is the same principle used for `ParallelRanking` (sec. 6.4.1) and places the load on a single peer. Multiple peers could be used as long as it is ensured that tuples with identical grouping values are routed to a common peer (performing grouping via routing), so all values for the aggregate are available. It is not possible to hash the values to a P-Grid key and route to the peer responsible because of P-Grid's replication (sec. 6.4.1). To work around this, a number of peers can be stored in the query plan by IP address and randomized hashing on the values can be used to generate an index into this list and route to the corresponding peer. This way, identical values will be processed on the same peer, and processing and network load will be balanced. Note that the variables can be swapped so that `ParallelExtract` is performed on x and `ParallelOIDMaterialize` on y.

It is also possible to implement grouping/aggregation using an extended version of `ParallelAVMaterialize`. This is useful when the plan contains few groups with many jtuples in each, which would lead to long processing in the first and message overhead in the second approach. Fig. 7.10 depicts an alternative query plan, first extracting the aggregation attribute y and then materializing x, so y data is shipped to the AV index peers for x, where the data is already clustered. `ParallelAVMaterialize` is extended by online processing as used in `ParallelRanking` and collects all jtuples received from the extraction peers to calculate the aggregate. Fig. 7.11 shows an example with two peers at each level: peer A sends the single `len` triple it stores to C and D. D discards it, but C is able to materialize `name`. B sends two jtuples to C and D. One can be used to complete the "E.T." group on C and calculate the aggregation on `len`. The same is true for peer D (in this simple example the group contains only one element).

It might seem easier to extract x and materialize on y, but this is not possible. The triples from peer C would be materialized on separate peers (A and B) because of a different distribution of the `len` AV index, thereby destroying the group.

7.6.2 Other New Operators

Integer Range Queries These queries are used to find data in a particular euclidian distance to a reference. They can be used to speed up integer $Nearest\ Neighbor - Rankings$ when a maximum distance has been specified, or for conjunctive `FILTER`

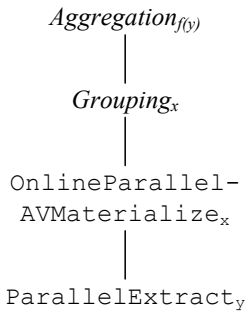

Figure 7.10: Aggregation with synchronization on the materialization peers

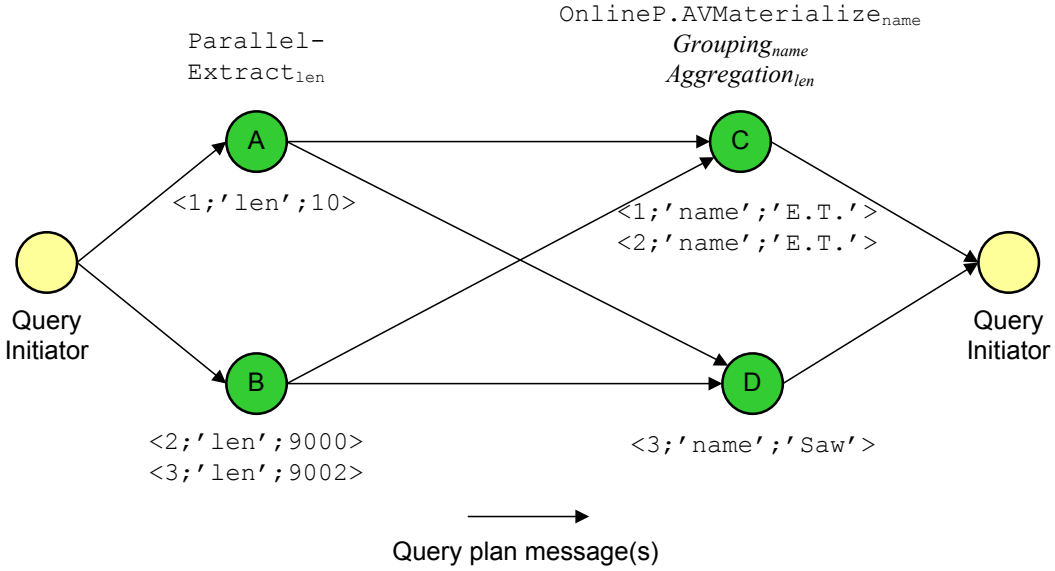

Figure 7.11: Example triple distribution (third approach)

clauses on the same variable which form ranges (for example, `FILTER x` $>$ `10 FILTER x` $<$ `20`). The hash function for integers maps them to their binary representation (limited to 31 bit). This makes it possible to use P-Grid range queries to extract all relevant data. Indexes which can be queried this way are the AV index and the Value index.

Substring Search q-gram indexes can be used for substring search. Only one q-gram from the substring needs to be hashed and looked up, the one with the lowest selectivity should be chosen. In this case, no post filtering must be applied. The existing `QgramExtract` and `ParallelQgramExtract` operators could handle this new operator with minor modifications. The VQL syntax must be extended to include this new operation.

SerialAVMaterialize The serial strategy only uses the OID index for $Materialization$. However, for a plan with many tuples the AV index might perform better. The OID index is usually distributed over more peers than the AV index, so shipping the plan to them takes longer and utilizes more bandwidth. Instead, the AV index should be used as for the `Extract` operator: the plan is sent to each of the peers storing data of the index, and the available triples are materialized. Only one attribute can be materialized at a time. Also, query plans which require the materialization of all attributes without specifying the names can not be instantiated with this operator.

Ship Where Needed Equijoin on Integer Data Similar to `ParallelQgramJoin`, which processes exact and similarity string joins, integer equijoins can be optimized. Instead of the AV similarity index the AV index must be accessed. Therefore, it is not possible to use `QgramExtract` internally, an operator similar to `Extract` would have to be used instead.

8 Implementation

This chapter documents the CouPé implementation. The basic architecture of the query processor has already been described in sec. 5.2. UML class diagrams will be used here for a more detailed view of the query planner, operators and execution engine. For each class, a short description will be provided. Furthermore, the integration of the query processor into the basic P-Grid P2P software and the statistics collector is documented. Note that the diagrams do not contain all existing relationships between the classes, only the relevant information is presented. Class and method names in the description are printed in `monospace`.

8.1 Implementation Details

True to the P2P approach, the implementation is present on all peers in the same form. Every peer accepts queries in VQL and can also process already running queries which get routed to it. The P-Grid source code is written in Java. The query processor is also implemented in Java and directly interfaces with P-Grid at the source level.

8.2 Query Planner

Fig. 8.1 gives an overview of the query planner, the classes will be reviewed in the following.

`PlanWrapper` Contains the query plan and associated information. Only the root operator is stored, but as each operator links to its children, the entire plan is referenced this way.

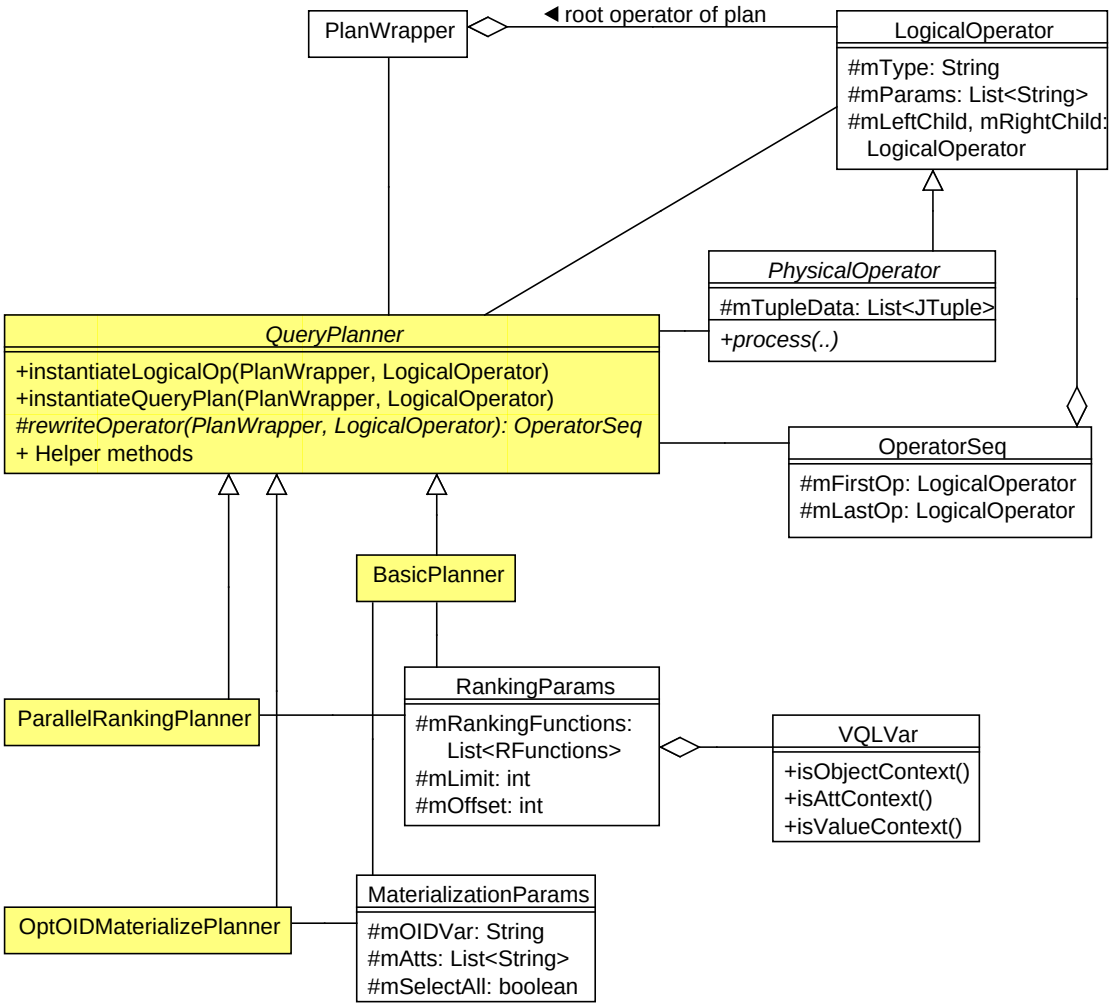

Figure 8.1: Query planner and related classes

`LogicalOperator` This class holds the information received from the VQL parser for every operator.

`PhysicalOperator` This is the base class for implementations of operators, and thus able to store jtuple data as required by the M^{2}QP concept. An abstract `process()` method is defined as well.

`QueryPlanner` Defines the basic interface for query planning. `instantiateLogicalOp()` replaces the specified `LogicalOperator` in the `PlanWrapper` with an implementation as determined by `rewriteOperator()`, which must be implemented by subclasses. `instantiateQueryPlan()` simply calls the former method for every operator in the plan.

`OperatorSeq` This helper class wraps a chain of linked operators in a class, used as the return type for `QueryPlanner.rewriteOperator()`, and makes it possible to replace n logical with m physical operators.

`BasicPlanner` This is the basic planner implementation described in sec. 7.4. It defines a mapping for every available logical operator.

`ParallelRankingPlanner, OptOIDMaterializePlanner` In contrast, only a mapping for $Ranking$/$Materialization$ is defined here.

`RankingParams, MaterializationParams` Helper classes which parse VQL parameter strings for $Ranking$ and $Materialization$, respectively, and provide get()-ter methods. This way, the functionality must only be implemented in one place and can be used by different planners.

`VQLVar` This is another helper class which parses the VQL variable notation and provides methods for often needed operations. It is used in many operators.

8.3 Operators

Fig. 8.2 shows some of the operator implementations available in CouPé and associated classes. One goal of the implementation was to avoid code duplication, so inheritance from other operators has been used where possible. All operators using it are depicted. `PhysicalOperator` is the base class of all operators and can store a list of

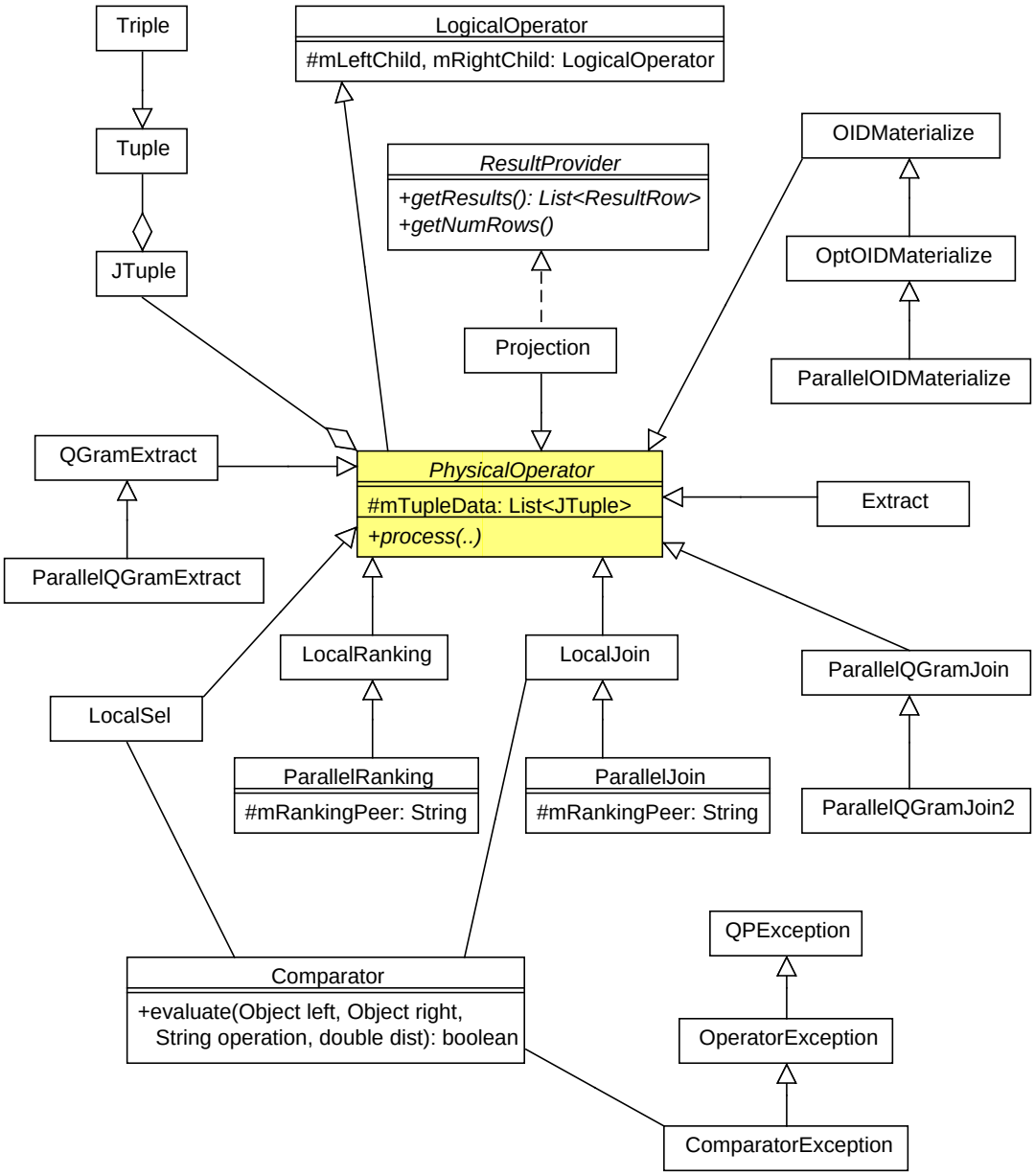

Figure 8.2: Operators and related classes

`JTuples`, equivalent to the data structure introduced in sec. 4.2.2. Each one stores multiple `Tuple` objects. Other relevant classes are:

`Triple` This special case of a `Tuple` is used in the `ExecutionEngine` and especially the operators.

`ResultProvider` This interface is implemented by operators which transform the data from the `JTuple` format to the final result – a table form with a column for every variable present in the `SELECT` statement. `Projection` is at the top of every plan created by the query planners, so it implements this interface. The `ExecutionEngine` at the query initiator can extract the results using the methods shown.

`Comparator` All comparison operations ($<$, $>$, $\leq$, $\geq$, $=$, $\neq$ and $\sim$) are performed in this class. It can handle Java `String`, `Integer` and `Float` data types, but only the first two were used for this work.

`QPException`, `OperatorException`, `ComparatorException` The exception hierarchy of CouPé.

8.4 Execution Engine

Fig. 8.3 shows the key component of the query processor, the `ExecutionEngine`. Important members of this class are:

`mOnlineData` The data for online processing operators (currently only `ParallelRanking`) is stored here if the peer is the synchronization peer. The map is indexed by the `GUID`[1] of the operator.

`mParallelJoinData` Similar to the previous structure, temporary data for parallel join processing is stored here.

`mQueryResults` Results of queries are collected here. `QueryResult` stores all relevant information for one query, including the `ResultProvider` (implemented by `Projection`) which holds the results and allows merging of two result query plans. The map is indexed by query ID (a `GUID`).

[1] globally unique identifier, this concrete implementation is part of P-Grid

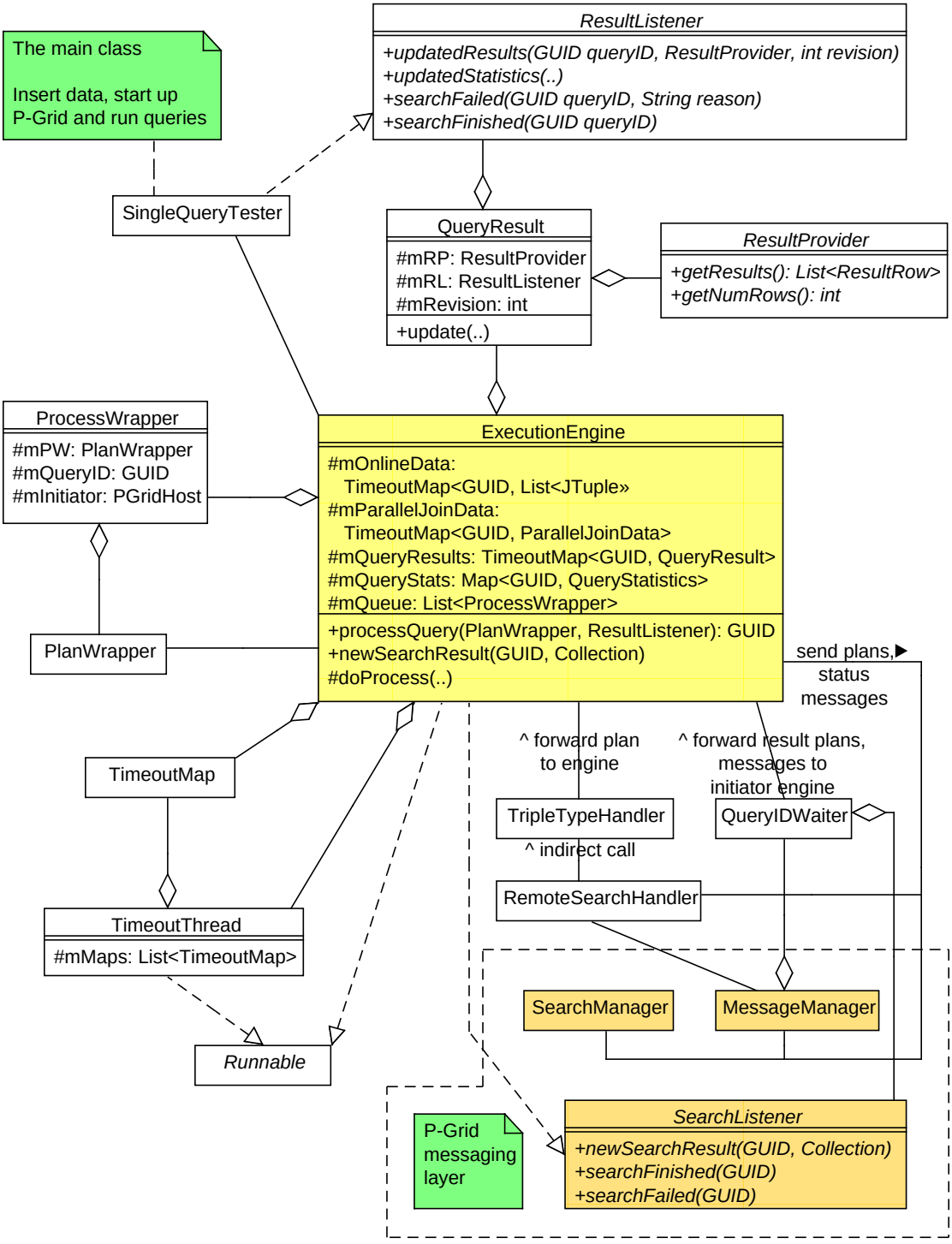

Figure 8.3: Execution engine and associated classes

`mQueryStats` Statistics for queries are aggregated, again indexed by the ID of the query.

`processQuery()` Applications call this method to start a new query. The query plan and a `ResultListener` callback must be provided. The query ID which is returned is used to identify the query in the callback methods.

`doProcess()` This is the main method and implements the algorithm presented in fig. 6.3. For sending query plans to other peers, the messaging layer of P-Grid is used (classes `SearchManager` and `MessageManager`). Results and status messages (`searchFailed()`, `searchFinished()`) are sent via `RemoteSearchHandler`, which is also responsible for forwarding incoming messages as received from the `MessageManager` to the `TripleTypeHandler`, which in turn passes them to the `ExecutionEngine`. The original implementation of this class is from the P-Grid codebase, but had to be changed to handle the new messages used for the transport of the query plans.

`ResultListener` is the callback interface to the application used to signal updated results, statistics and query status.

`mQueue` Query plans to be processed are queued up here. This includes both locally started queries and those received from other peers for further processing. A thread fetches queries and executes them asynchronously by calling `doProcess()`.

`ProcessWrapper` is a helper class for the processing queue and holds information about the queries scheduled for processing.

`QueryIDWaiter` A callback used on the query initiator to receive query results, status messages and heartbeats pertaining to a specific query ID. The messages are signalled to the `ExecutionEngine` by the P-Grid `SearchListener` interface, which is implemented by the `ExecutionEngine`.

`TimeoutMap`, `TimeoutThread` Many map data structures in the engine are backed by a `TimeoutMap` which discards entries after a period without `get()` or `put()` access, freeing resources of queries which have failed or will not provide any more results.

`SingleQueryTester` is used for the tests in sec. 9. All parameters, including P-Grid settings like replication factor, bootstrap hosts and running time are configured here. It starts up P-Grid on the peer, inserts the specified data and joins the network. A command line interface is available for starting queries and fine-tuning all aspects of the `ExecutionEngine`. It is also possible to run the tester in the background and send commands to a fifo file. This makes it possible to run tests from one master host which

sends commands to many different peers running the tester with minimal manual intervention. `SingleQueryTester` implements the `ResultListener` which will be called from the `ExecutionEngine` for the queries started on that peer.

8.5 P-Grid Integration

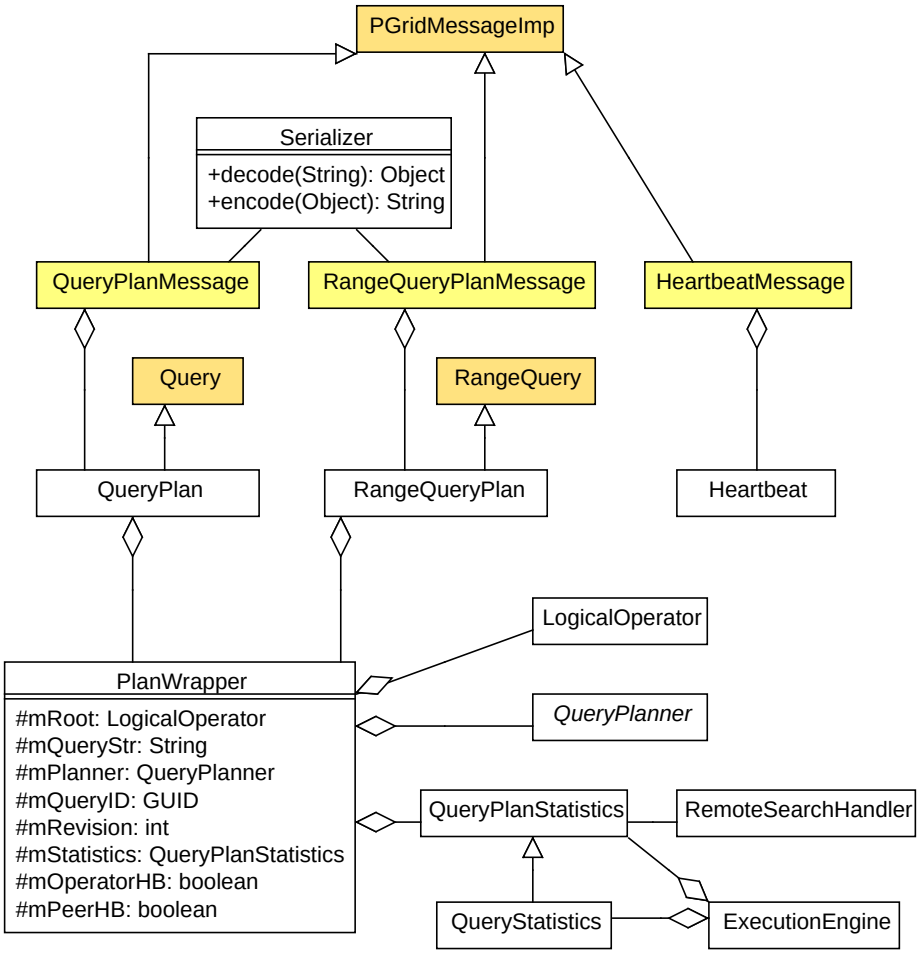

Figure 8.4: Implemented messages and P-Grid base classes

During development of CouPé, three different P-Grid versions were released. The integration with the latest is documented in fig. 8.4. Implemented messages and P-Grid

classes are highlighted (yellow and orange, respectively). P-Grid messages are XML data which are compressed for transmission. Therefore, data must be transformed to an appropriate string serialization. For the nested structures encountered in query plans this can be cumbersome. Also, due to the nature of a research project, no exact requirements were known, leading to constant change. It is very time-consuming to alter the message each time data structures change. Therefore, the `PlanWrapper` was designed as a class to hold all relevant data for a query. This single object is serialized to a byte stream, encoded as Base64 and included in the XML message. This process is implemented by the `Serializer`. For this to work, all classes contained in the `PlanWrapper` must implement the Java `Serializable` interface, but as it is an empty interface, this is only a declaration. P-Grid already offers the basic classes `QueryMessage` and `RangeQueryMessage`. They encapsulate an `Query` and `RangeQuery` object, respectively, which contains all relevant information stored in the message, and construct the XML representation from that. The query processor implements two new messages, `QueryPlanMessage` and `RangeQueryPlanMessage`, in the same way: `QueryPlan` and `RangeQueryPlan` are subclasses of the P-Grid containers and both aggregate one `PlanWrapper` object to hold the additional query plan information. The same code for message construction is used with the addition of a new tag for the encoded `PlanWrapper` data.

For the heartbeat message, only a single string was used as payload, so no `PlanWrapper` is used in this case. The string is simply attached to the message with an additional XML tag.

In fig. 8.4, the `PlanWrapper` is displayed in detail. As shown previously, it contains the reference to the root plan operator (and therefore to the complete plan). `mQueryStr` is the original VQL query, `mOperatorHB` and `mPeerHB` control the sending of heartbeats on a per-plan basis. `mPlanner` can refer to any planner implementation and will be used during processing if the plan contains logical operators. An important part of each query are the statistics, which are essential for a thorough evaluation. They are collected and updated during processing of the query in `QueryPlanStatistics` objects and maintained in the plan. This is performed in `RemoteSearchHandler` and `ExecutionEngine`, as all messages will pass through these classes. `QueryStatistics` aggregates all statistics for one query provided by multiple result plans on the initiator. Some of the statistics collected are the number of hops and messages for a plan and the bandwidth used. They will be used for the evaluation of the query processor in sec. 9. The statistic component was implemented by Marcel Karnstedt.

UML diagram	Source Code
ExecutionEngine	QueryProcessor
QueryPlanner	PlanRewriter
BasicPlanner	SimplePlanRewriter
JTuple	NamespaceTuple
RankingParams	PhiParams
MaterializationParams	OmegaParams
mOidVar	mNamespace

Table 8.1: Differing variable and class names in the source code

8.6 Implementation Class and Variable Names

Some of the names used in the UML diagram and the rest of this work differ from the ones used in the source code. This was done for better understanding and consistency. The changes are documented in table 8.1.

9 Evaluation

This chapter analyzes the tests that were performed to evaluate the query processor. First, the key questions to be answered by this chapter are put forward. The test setup and the captured statistics are documented, followed by a discussion of each test. The last section summarizes the most important findings.

9.1 Introduction

The key questions to be answered by the evaluation are:

1. Is query processing in a DHT feasible?
   - Is the quality of search results acceptable?
2. Does the query processor scale gracefully?
3. How do different operator implementations compare against eachother? Which should be used in a given situation?

First, rudimentary tests in a LAN with up to 35 peers were performed, which were successful. In the second step, CouPé was deployed on PlanetLab [CCR+03].

9.1.1 PlanetLab

PlanetLab is a global research network consisting of more than 700 nodes worldwide located at universities and research institutions. The idea is that participants contribute a few nodes at their site and are granted the privilege to run tests on all nodes in the network in turn. This makes it possible to deploy and test large-scale distributed services without a prohibitively high initial investment. PlanetLab provides an ideal environment for testing CouPé in a WAN setting. It is also a challenging environment

where peer failures are to be expected and few guarantees can be made about the available resources on a node. Therefore, it also provides insight whether CouPé is able to function in such adverse situations.

Initial tests on PlanetLab were performed with up to 100 peers and remaining problems identified and removed. For example, in a WAN, the transmission of messages can take longer due to low bandwidth links and high latency. This led to timeouts and frequent retransmissions which were fixed by increasing P-Grid's internal timeouts and implementing additional message queues. The limited resources caused by the many experiments running in parallel on each node had also to be taken into account.

9.2 Test Setup

A set of 119 test queries were created to test the various operators. Four different P-Grid nets were constructed from scratch, consisting of 30, 60, 90 and 120 peers. Each peer inserted 5 out of 600 total tuples, so the smallest net contained 150, the largest 600 tuples. This way, the total data in the net grows, but the data each peer must keep in its data store (which is not identical to its inserted data) stays constant. There are four different schemas for the tuples. They are identical except for the second column, which stores string data, but has different names: for 300 tuples it is `na`, for the other 300 `title`, and 16 tuples have been modified in each set to contain a slighty different name (`na` and `titel`), analog to heterogenous schemas from different participants, which makes it possible to test similarity selections and joins on schema level. The other columns contain integer data. The tuples inserted by each peer are chosen so that each net size has an equal percentages of tuples from each schema.

`A(nr, na, len, count, date, r)` (16 tuples)
`B(nr, n, len, count, date, r)` (284 tuples)
`C(nr, title, len, count, date, r)` (284 tuples)
`D(nr, titel, len, count, date, r)` (16 tuples)

The reason for the relatively low number of tuples per peer is that memory problems on the PlanetLab nodes prevented larger setups. Later tests revealed that this was caused by problems in P-Grid which were fixed by an update, but it was not possible to re-run all tests in time.

The following indexes (sec. 4.1) were created:

- OID index
- AV index
- AV similarity index (second column)
- Attribute similarity index on columns `nr`, `title/titel/na/n`, `len`, `r`

Table 9.1 shows the average number of triples generated from these 5 tuples per peer.

Index	Triples/Peer
OID	30
AV	30
AV sim.	55.3
Att. sim.	85.8
Total	201.1

Table 9.1: Average number of triples per peer and index

The queries were executed and statistics gathered in log files. The problems discovered during the first run led to a second run using an updated P-Grid version. Most of the queries had to be repeated. Statistics from the first run are only used by the materialization test (sec. 9.4.3), the quality was good here. As expected, some nodes dropped out of the network while the queries were executed. This was mainly caused by large memory consumption and Java exceptions, but in the second run much less problems occured. The most failures occured in the 90 peer net for this run (7 failures, 7.8%). P-Grid's replication feature was used (replication factor 2), so these moderate failure rates could be tolerated.

9.3 Statistics

Extensive statistics are collected on a per-query basis (sec. 8.5). They can be parsed and examined with LogAnalyzer, a tool implemented for that purpose by Marcel Karnstedt. Some of the collected statistics, which will be used for analysis in the remainder of this chapter, are:

Query Plan Hops The number of times a query plan was routed by the query processor in succession. For parallelized plans, the maximum over all instances is used. This gives a good indication of the total time needed to process the query, as the dominating factor for this is the transmission of messages between peers. For example, when the first processed operator in a plan is `ParallelExtract` and routed to three peers in parallel, this is counted as one hop (and 3 query plan messages, see below). Assuming one of the plans is routed 3 more times before returning results and the other two one more time, the hop count for the query is $1 + 3 = 4$. The number of query plan messages is $4 + 2 + 2 = 8$.

Query Plan Messages The number of times a plan was routed by the processor. For parallel plans, the sum over all instances is calculated. This is a key statistic for analyzing the message load in the network.

P-Grid Messages The number of times a query plan was transmitted between two peers as result of a routing request from the processor. Query plans are routed to P-Grid paths, but each route can consist of multiple hops in the underlying network, which are counted here.

P-Grid Hops Similar to query plan hops this records the maximum number of hops for all concurrently routed plans on the level of the underlying (IP) network.

Bandwidth The bandwidth required for routing of the compressed plan(s) during processing, including message headers, but excluding the result plans.

Reply Bandwidth The bandwidth required for sending the compressed result plan(s) to the initiator.

For even more detailed analysis, logical and physical versions of most of these statistics are kept. When a plan is scheduled for routing but P-Grid detects that it can be handled on the local peer this is counted as a logical hop and message only. Physical statistics are used for analysis in most cases so that the advantages of operators (i.e., better routing) become visible. This requires repetition of queries and averaging of statistics, so random effects can be reduced. During the first run, each query was repeated 3 times, during the second 5 times. To prevent overlap in query execution, waiting periods were used in between. Because of this, the number of repetitions could not be set higher.

9.4 Tests

9.4.1 P-Grid Network

First, basic P-Grid statistics will be presented. Fig. 9.1 shows the average and maximum path lengths for the four net sizes. Both show moderate growth. The maximum path size does not exhibit any extremes, which implies that the virtual P-Grid tree is not heavily skewed.

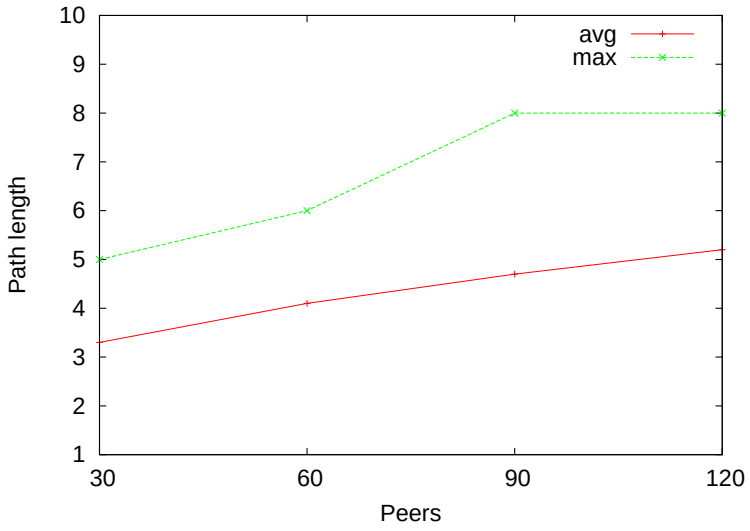

Figure 9.1: Average and maximum P-Grid path lengths

An important characteristic of a DHT is the number of hops required to retrieve a key ("lookup"). Less hops imply less waiting time. P-Grid guarantees logarithmic costs w.r.t. the number of key space partitions, and fig. 9.2 shows that this is the case in practice.

9.4.2 Extraction

The VQL query `SELECT c WHERE { <x;'n';c> }` was used to compare the two *Extraction* implementations `Extract` and `ParallelExtract`. The resulting query plans contain an additional `Projection` operator at the root. Attribute `n` was chosen because it hashes to a relatively short P-Grid key, so its AV index is stored on more than one

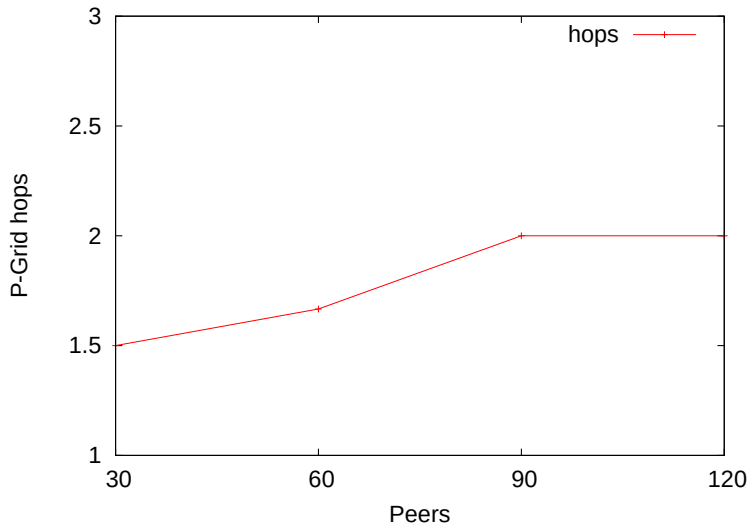

Figure 9.2: P-Grid hops for key lookup

peer with high probability which emphasizes the differences between the two implementations.

Fig. 9.3 shows that both plans require a similar number of P-Grid messages to retrieve all data. Linear growth can be observed, as the number of AV index peers also increases with net size. The plot for P-Grid hops (fig. 9.4) shows the key difference between the operators: while `Extract` requires as many hops as messages because it contacts the peers in sequence, P-Grid's prefix queries speed up `ParallelExtract` considerably.

9.4.3 Materialization

Four different *Materialization* operators exist: `OIDMaterialize`, `OptOIDMaterialize`, `ParallelAVMaterialize` and `ParallelOIDMaterialize`. To compare them, the query plan depicted in fig. 9.5 was constructed. The `Extract` operator is in the DONE state and contains 5 triples[1], chosen randomly from all inserted triples of the `len` attribute. They will be extended by two further attributes by *Materialization*. Fig. 9.6 shows the physical query plan hops for each operator. `OIDMaterialize` routes the plan for each

[1] or more generally, jtuples

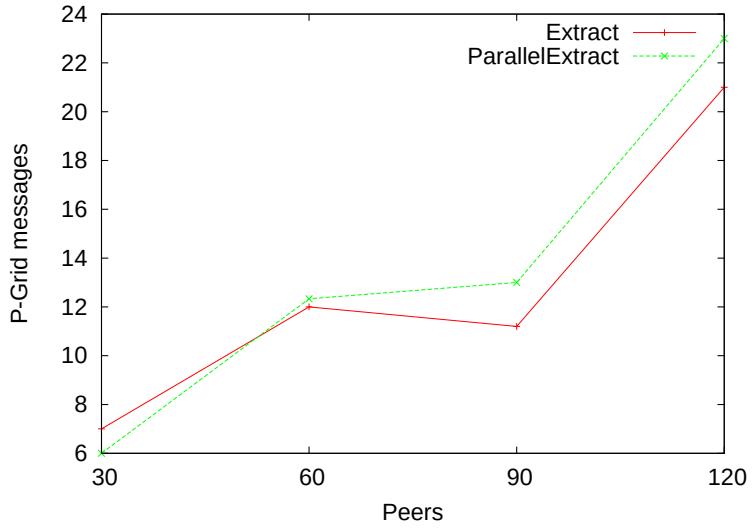

Figure 9.3: Extraction: P-Grid messages

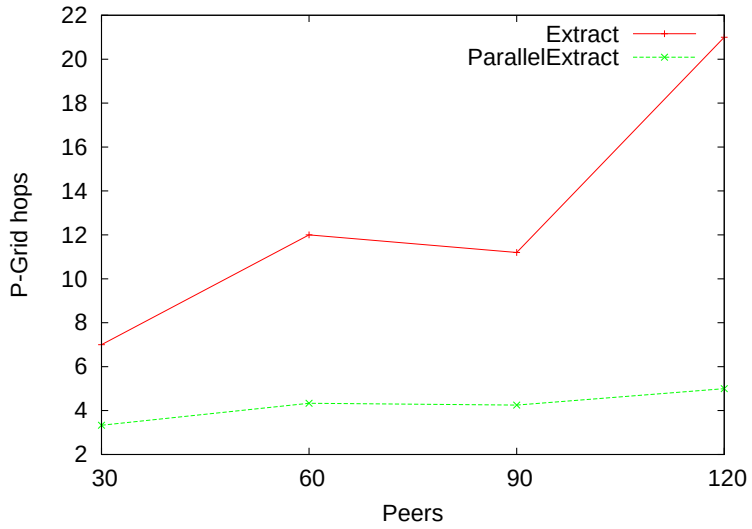

Figure 9.4: Extraction: P-Grid hops

jtuple, so the expected hop number is 5. As the physical hop number is plotted, which does not increase the hop count when the plan can be processed on the local peer, this number is below 5 for two of the nets. As expected, `OptOIDMaterialize` performs better except for one net. `ParallelAVMaterialize` requires a constant two hops. Because it can process only one attribute, two instances of the operator are used in succession for this query and each requires one query plan hop to reach the peers storing the corresponding AV index. `ParallelOIDMaterialize` generates and routes a plan for each jtuple simultaneously, so it also requires one hop. Because the OID index is used, the number of attributes to be materialized is not relevant. For `OIDMaterialize` and `OptOIDMaterialize`, a slight rise in the number of hops can be seen for increasing net size. This is because the average path length grows while the length of the requested keys stays constant, so the probability that the local peer does not store the requested key increases. With increasing net size the query plan hops required will converge towards the number of jtuples in the plan and can be predicted precisely.

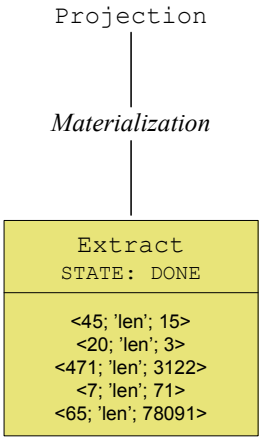

Figure 9.5: Materialization: query plan

Fig. 9.7 shows the transmitted query plan messages for each operator and highlights the difference between the three OID index operators and `ParallelAVMaterialize`. The number of messages routed in parallel by `ParallelOIDMaterialize` are similar to the other OID operators. The `ParallelAVMaterialize` plan requires a constant two messages, and analysis of the logs show that the AV indexes for the materialized attributes were both stored on one peer only, so one message for each was sufficient to route the plan. This distribution puts the operator at an advantage, and more tests should be performed to analyze the properties for distributed AV indexes in more detail.

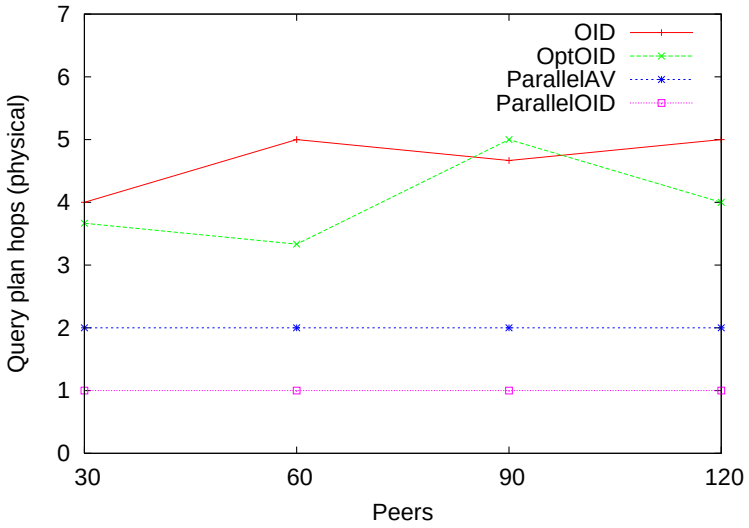

Figure 9.6: Materialization: query plan hops

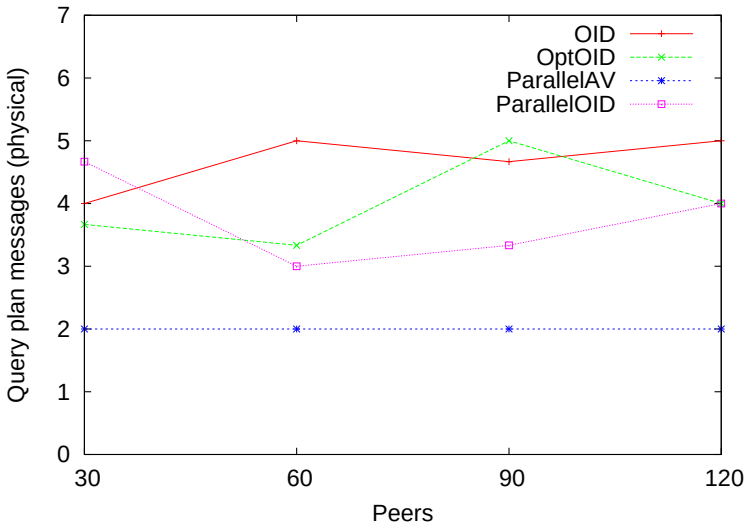

Figure 9.7: Materialization: query plan messages

Bandwidth consumption during processing is shown in fig. 9.8 and correlates with the number of messages. `ParallelAVMaterialize` comes out on top, as it only needs to ship the plan two times. The first time it contains the 5 jtuples to be materialized, the second time also the first materialized attribute. Between the three OID operators `ParallelOIDMaterialize` performs best. It only ships the plan with one of the 5 jtuples stored in the `Extract` operator 5 times, while the other two have to transmit all 5 jtuples along with the accumulating materialization results. An improvement would be the implementation of a consumer interface instead of the current cursor model used for iterating over jtuples stored in child operators, which would reduce the size of the plan when operators only need to access the data once. `OptOIDMaterialize` yields very different results for the net sizes. This is most likely caused by the low number of repetitions (3) used for the averages in conjunction with the mode of operation. For example, when 4 out of the 5 jtuples in the plan can be materialized on one peer, it is much more efficient to materialize the single tuple first, so less data will have to be transmitted to the peer where plan execution can be completed. This optimization should be implemented in the operator.

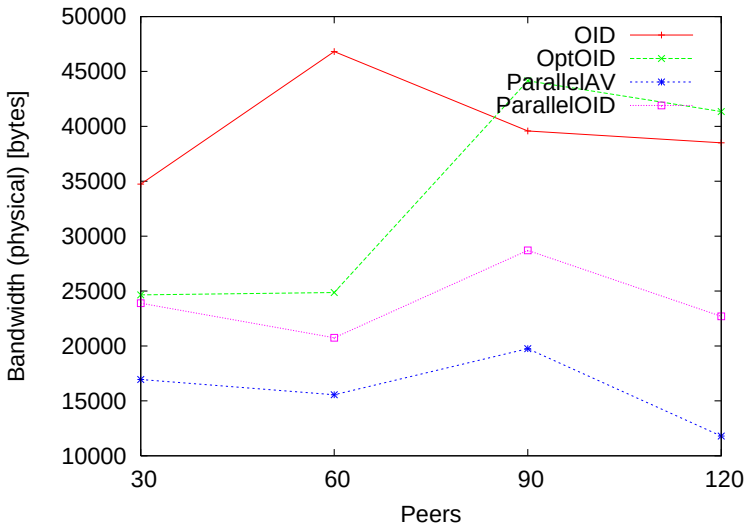

Figure 9.8: Materialization: query plan bandwidth

Because `ParallelOIDMaterialize` must send 5 result plans to the initiator, it has the highest reply bandwidth consumption (fig. 9.9) – it grows linearly with the number of jtuples to be materialized. The peak for the smallest net corresponds to the peak number of messages sent as seen in fig. 9.7. When the initiator cannot process one of the plans generated by the operator it routes it, causing an additonal physical message and

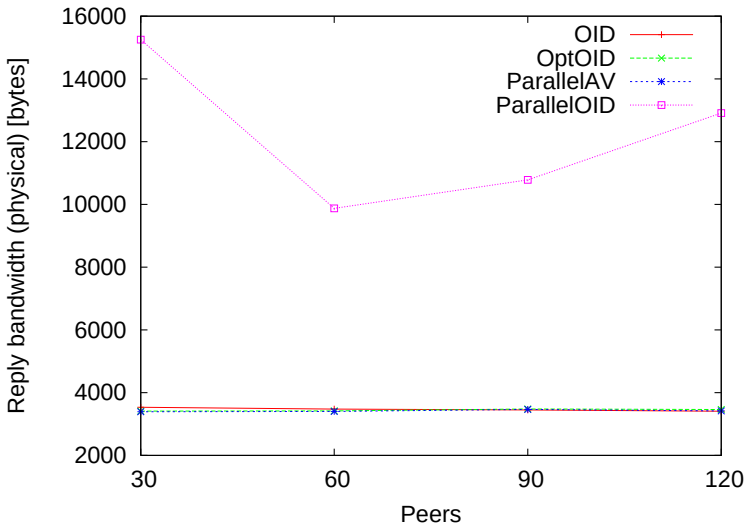

Figure 9.9: Materialization: reply bandwidth

Plan	Attributes
1 att.	count
2 att.	count, nr
4 att.	count, nr, date, r

Table 9.2: Materialization attributes

also an additional reply message, which would otherwise be logical only. It is assumed that the low number of repetitions combined with a disadvantageous sample of initiator peers caused this skew. The other three operators only ship one plan of the same size to the initiator[2]. Replies in P-Grid are sent over a direct connection to the initiator at IP level, so no additional hops are required, which can be seen by the constant bandwidth consumption for increasing net size.

Further tests where done materializing 1 and 4 attributes (table 9.2). The OID index operators are hardly affected by this, but it is interesting how `ParallelAVMaterialize` handles the situation, as each attribute must be processed by a separate operator. Fig. 9.10 shows the bandwidth consumption, including `ParallelOIDMaterialize` on 4

[2] this would have been different had the AV index of the two attributes been stored on more than one peer each

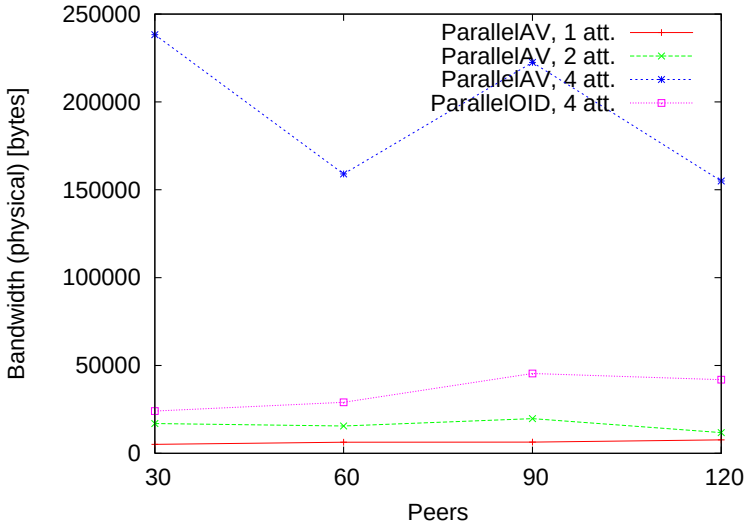

Figure 9.10: Materialization: different number of attributes

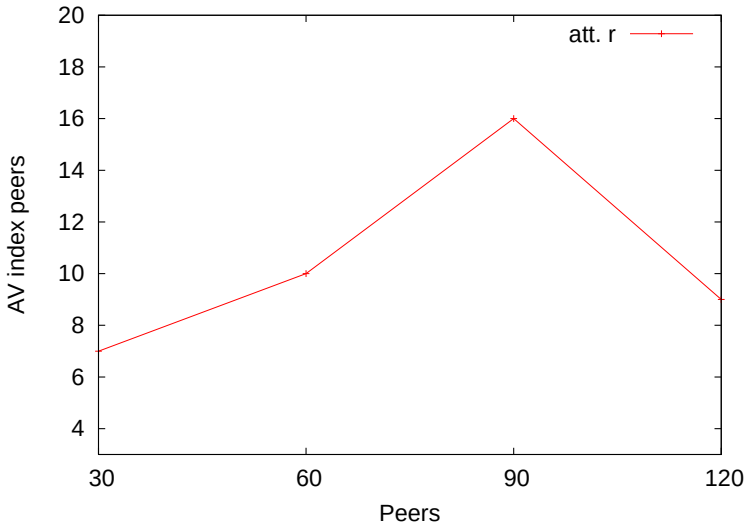

Figure 9.11: Materialization: AV index peers, att. `r`

attributes for comparison. Only one of the four attributes, `r`, has an AV index spanning multiple peers. It is only materialized in one query and leads to a dramatic increase in bandwidth, because the plan must be multicasted to all AV index peers. The number of peers for the attribute is plotted in fig. 9.11, and except for the smallest net size they show correlation with the bandwidth plot (log file analysis did not show any reason for the anomaly). The plot also shows the overhead of `ParallelAVMaterialize` as pointed out in sec. 6.4.1: Although only 5 jtuples need to be materialized, the complete plan must be sent to every AV index peer, even though there are more than 5 for all net sizes, so some of them will not contribute any results.

These materialization tests show that three key parameters should be considered by an optimizer to determine the best operator for a given query:

- number of attributes
- number of jtuples
- number of peers storing the AV indexes

9.4.4 Similarity Selection

Similarity selection on strings is a key feature of CouPé. The following VQL query was used for evaluation:

```
SELECT v WHERE { <x;'n';v> FILTER v ~ 'prize_the', 4 }
```

It is rewritten to the query plan shown in fig. 9.12, and for $Extraction_n$, one of these physical operators is used: `Extract`, `ParallelExtract`, `QgramExtract`, `Parallel−QgramExtract`. The first two simply access the AV index of attribute `n` and extract all data. The serial plan is at a slight disadvantage here, as it cannot filter until all data has been extracted, while the parallel version can process `LocalSelection` on each index peer in parallel and does not have to ship accumulating `Extract` results from peer to peer. The q-gram operators need to look up 5 non-overlapping q-grams in the AV similarity index to answer a query with distance 4. Because the search string is too short for this, all overlapping q-grams (11) are generated instead. After hashing, 9 distinct P-Grid keys remain which are used for routing. The query selects 4% of the `n` column.

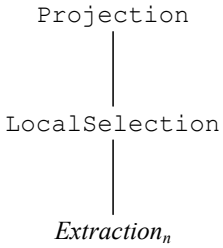

Figure 9.12: Similarity selection query plan

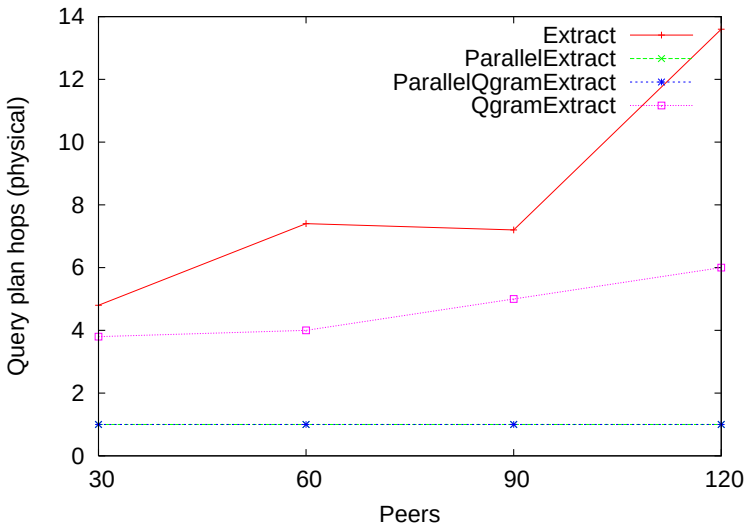

Figure 9.13: Similarity selection: query plan hops

Query plan hops taken are depicted in fig. 9.13. As expected, the parallel operators require one hop. `QgramExtract` takes fewer than the expected 9 worst-case hops, because physical hops are plotted. With growing net size this number slightly increases, as the probability that a q-gram lookup can be answered without routing becomes lower (sec. 9.4.3). `Extract` clearly suffers from the number of AV index peers (proportional to net size), because it has to fetch all the data of the attribute regardless of the queried distance.

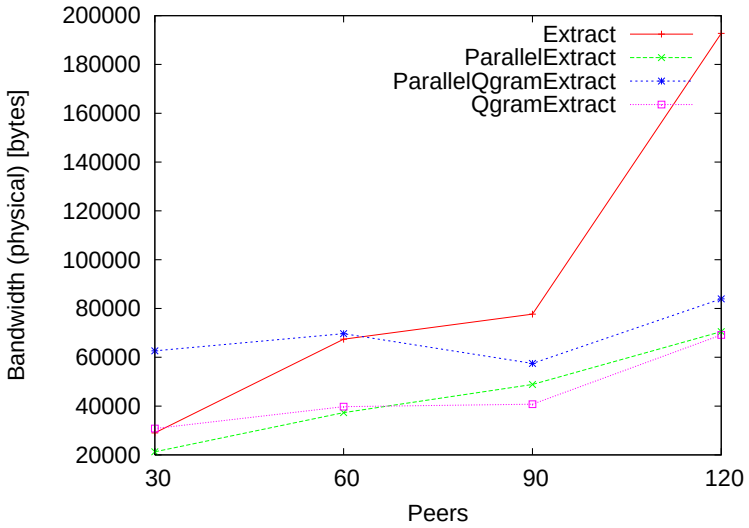

Figure 9.14: Similarity selection: bandwidth

Fig. 9.14 plots the bandwidth consumed by the plans. `QgramExtract` performs better than the parallel version even though it accumulates the temporary results in the query plan during routing. One reason is that `ParallelQgramExtract` has to ship a message for each of the distinct q-gram keys, even if they are stored on the same peer. `QgramExtract` only routes the plan once to such peers. The overhead of many messages is noticable because the result set is relatively small. For a bigger edit distance, `QgramExtract` would have to ship more intermediate results between peers and would perform worse. `ParallelExtract` only ships the query plan without any intermediate results to all peers storing the AV index. As the net size increases, more peers must be contacted and bandwidth consumption grows linearly. `Extract` shows exponential growth because the intermediate data and the number of hosts routed to in sequence both increase with net size.

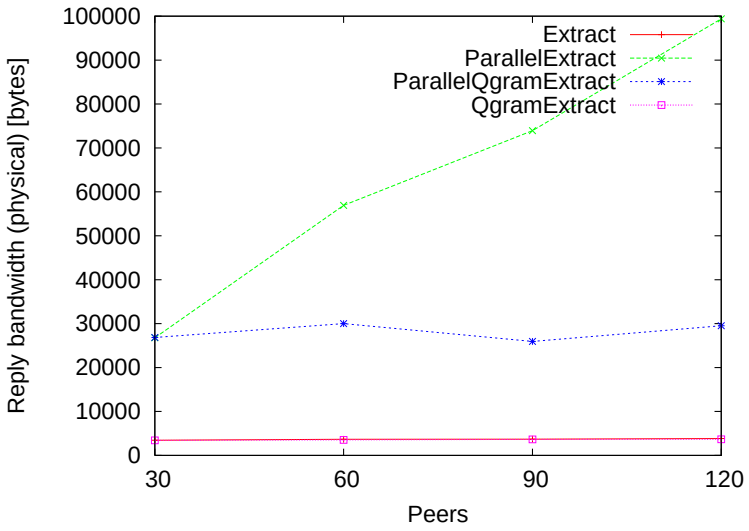

Figure 9.15: Similarity selection: reply bandwidth

Again, the serial operators perform best when it comes to reply bandwidth, as all results are contained in a single plan (fig. 9.15). Although the larger nets contain more matches this is not noticable, as plan overhead dominates the relatively small result sets, and the consumed bandwidth nearly stays constant. `ParallelQgramExtract` performance correlates with the constant number of parallel plans (one for every distinct q-gram key). As the number of AV index peers increases, the reply bandwidth for `ParallelExtract` grows linearly. The relatively small result payload is clearly dominated by the overhead of the query plan and P-Grid message data.

Important parameters for consideration by a future optimizer for similarity selection are:

- distance
- number of AV index peers
- selectivity

`Extract` is outperformed by `QgramExtract` in most cases and is only useful if no q-gram indexes can be created. Depending on the requirements, `QgramExtract` (low reply bandwidth) or one of the parallel operators (low latency) should be used. In the second case, `ParallelQgramExtract` is a good fit for small distances while `ParallelExtract` performs good for few AV index peers.

9.4.5 Similarity Join

The following VQL query was used for evaluation of similarity joins on string data:

```
SELECT xn,yn WHERE { <x;'titel';xn> <y;'n';yn> FILTER xn~yn, 3}
```

Five different physical plans were tested:

1. `ParallelExtract` operators fetch the data and the join is processed in parallel on the peers storing the AV index of the right-side attribute `n` (fig. 9.16).
2. The left side is processed by `Extract`, the right by `ParallelQgramJoin` (sec. 7.3.2). Because the left-side attribute `titel` contains relatively few tuples, this ship-where-needed-approach becomes interesting (fig. 9.17).
3. Like (2), but on the left side, `ParallelExtract` is used. This parallelizes processing at this stage and also for `ParallelQgramJoin`, which can generate plan cloning messages on multiple peers in parallel and thus balance this load.
4. Like (2), but using `ParallelQgramJoin2` on the right.
5. Like (3), but using `ParallelQgramJoin2` on the right.

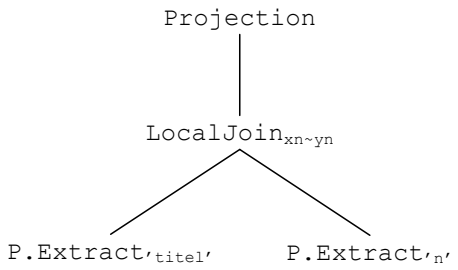

Figure 9.16: Similarity join: `ParallelExtract`

Table 9.3 summarizes the operators on the left and right side of the join for each variant and the planners used. The rest of the plan (i.e., `LocalJoin` and `Projection`) is identical. The query result size is approximately 2% in relation to the total number of tuples (for the smallest net with 150 tuples, 3 results are generated). Fig. 9.18 depicts the bandwidth. Plans 2/3 and 4/5 perform very similarly. Analysis of the log files shows that the AV index for attribute `titel` was only stored on one peer in each net, so the expected worse performance for the serial versions (2 and 4) does not become visible.

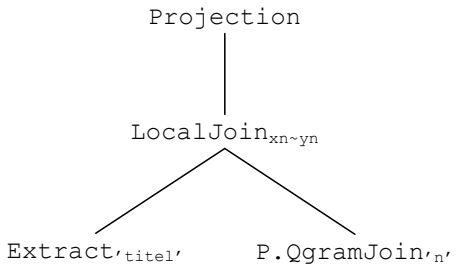

Figure 9.17: Similarity join: `ParallelQgramJoin`

Plan	Left	Right	Planners used
1	`ParallelExtract`	`ParallelExtract`	P.Extract, Basic
2	`Extract`	`ParallelQgramJoin`	P.QgramJoin, Basic
3	`ParallelExtract`	`ParallelQgramJoin`	P.Extract, P.QgramJoin, Basic
4	`Extract`	`ParallelQgramJoin2`	P.QgramJoin2, Basic
5	`ParallelExtract`	`ParallelQgramJoin2`	P.Extract, P.QgramJoin2, Basic

Table 9.3: Similarity join: Tested query plans

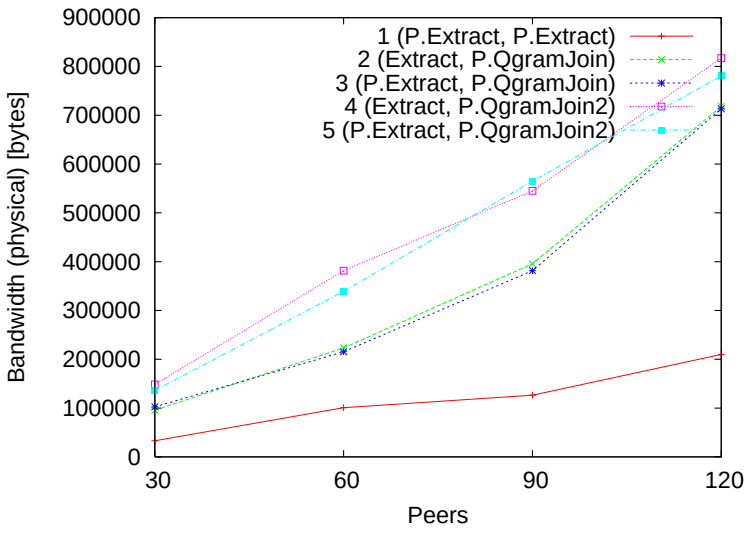

Figure 9.18: Similarity join: bandwidth

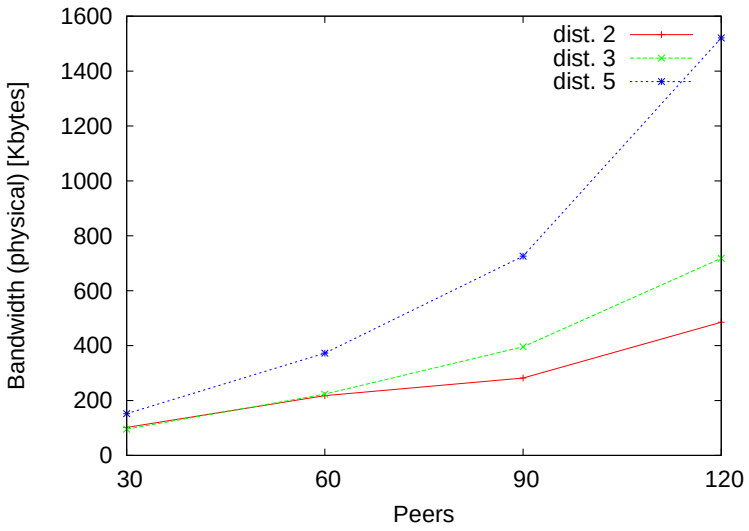

Figure 9.19: Similarity join: bandwidth for different edit distances

Plans 1, 4 and 5 show linear performance and thus scale with the result size. The serial q-gram operators used internally by plans 2 and 3 do not seem to scale very well. As they are routed to all the q-gram keys in sequence, the results in the plan accumulate and lead to exponential bandwidth consumption. This is further illustrated in fig. 9.19, which compares the second physical plan for three different distances (2, 3 and 5). With growing distance, more temporary results accumulate in the plan and must be routed from peer to peer. Still, plans 4 and 5 require more bandwidth for the net sizes shown in fig. 9.18, which is caused by the overhead of the higher number of messages generated by the internally used `ParallelQgramExtract` operators. Plan 1 performs best: it only has to ship few tuples from the lone left-side AV index peer to each AV index peer on the right side, where the results are processed and replied to the initiator without further routing. No intermediate results are sent around the network.

The difference between the two `ParallelQgramJoin` operators also becomes clear when looking at the query plan hops (fig. 9.20). One hop is needed by all plans to extract the data of the left-side attribute. For plan 1, another is required to route to the other side. Plans 4 and 5 use `ParallelQgramExtract` internally which routes q-grams in parallel, registering one additional query plan hop. Plans 2 and 3 make use of `QgramExtract` operators, which route to the q-grams in sequence and need four additional hops (for a edit distance of 3, 4 q-grams are generated for each jtuple of the left side). Because physical hops are plotted, the number is lower than the theoretical

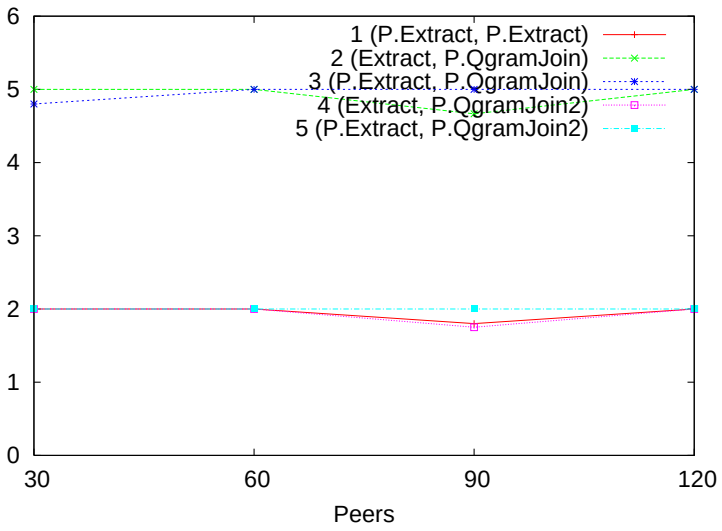

Figure 9.20: Similarity join: query plan hops

upper bound in some cases.

As for similarity selection, the operators generating fewer plans require much less reply bandwidth (fig. 9.21). Because of the small result size, much of the bandwidth is consumed by message overhead.

Overall, plan 1 performs well in most areas. It consumes the least total bandwidth (also including reply bandwidth) and has low latency. The main reason is the structure of the data. On the left side, there are relatively few data items. Shipping them to the AV index peers of the right-side attribute does not require much bandwidth. `ParallelQgramJoin` groups identical values from the left side together and generates a `QgramExtract` query plan for each (sec. 7.3.2). As the data items are distinct in this case (they are film titles), many messages must be generated. The operator cannot play to its strengths and the overhead of each message leads to high bandwidth consumption. When these parameters change, the specialized similarity join operators should perform much better.

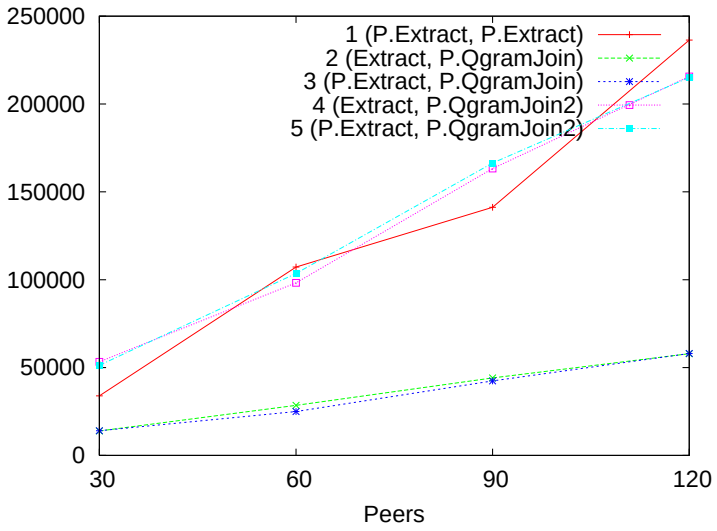

Figure 9.21: Similarity join: reply bandwidth

9.4.6 Schema Similarity Queries

Schema similarity queries (selections and joins) were also tested. However, in none of the runs could good statistics be obtained. The log files identify two of the key problems for this:

- The same operators as for instance similarity queries were used, but schema queries produce larger result sets and temporary results in general. They caused problems for serial operators like `QgramExtract`, so it was not possible to finish queries using those operators in the two biggest nets. In a WAN, large plans are much more likely to cause timeouts and serial operators produce such plans quickly for schema operations.
- As scalability problems for the serial operators were anticipated, most of the remaining plans used prefix query operators to process large amounts of data in a distributed fashion. While they worked correctly for the instance level tests discussed above, problems appeared when they were applied to an empty P-Grid key prefix, which is needed on schema level. Queries located only part of the results in this case. This is similar to a bug which occured during the tests of the first run and was fixed with a P-Grid update, so it is expected to be removed in an upcoming version and the schema tests should be repeated. Such queries also

worked with earlier P-Grid versions.

9.5 Summary

The tests in this chapter show that complex query processing on top of P-Grid is feasible and performs well in many areas. Except for the schema similarity queries (sec. 9.4.6), most results were complete, even though peer failures occured. P-Grid's replication and routing layer perform as expected. Operators for the most important operations were analyzed, including key lookup, $Extraction$, $Materialization$, similarity selection and similarity join. Many physical operators show good scalability properties. Some implementations are a good fit when low latency is required, others consume few bandwidth. There also exist implementations which are not suited for a large-scale environment because they do not scale well with net size. The gathered information could be used as groundwork for an (adaptive) optimizer. It was not possible to analyze schema operations, but this could be addressed by future work.

10 Conclusions and Outlook

10.1 Conclusions

The main objectives of this thesis were the development of a distributed query processor built on top of P-Grid and its evaluation. The Mutant Query Plan concept was chosen as the basic model for query execution. The logical operators of the algebra were implemented. Query planners which build physical query plans from the plans generated by the VQL parser were created. The execution engine is the main component on each peer. It interfaces with the P-Grid messaging layer, receives query plans, processes the operators in them in the correct order, routes them to other peers and sends the results back to the initiator. This architecture works well: even without any central components, queries are processed reliably and correctly.

The original MQP concept is a serial approach and relatively slow, so M^2QPs were introduced. They parallelize plan execution by using prefix queries, plan cloning and parallel execution of branches of binary operators. This can lead to great speedups, which was confirmed by the evaluation. Synchronization peers are used for correct processing of blocking operators for parallelized queries. This is not ideal, as a central component is not desired in a P2P setting, but an optimal solution for this was not in the scope of this work. Similarity operators for selection and join were presented and evaluated. For increased efficiency they make use of special q-gram indexes. Like the majority of the implemented operators they can process data on schema level.

CouPé was evaluated on up to 120 PlanetLab nodes. The experiments demonstrated the feasibility of the concept and performance characteristics for some of the implemented operators could be obtained. Except for the schema similarity queries, which faced several obstacles, most of the queries provided complete results.

P-Grid proves to be a good DHT choice as it offers prefix queries and replication. The former provide an efficient way to disseminate query plans in parallel to all peers which need to process them, the latter ensures availability of data in the face of peer failures, which have to be expected in large-scale WAN settings. The partitioning scheme and

the indexes created make it possible to efficiently execute operators on the data. In some cases, different indexes can be used to implement a logical operation, which can be very useful for a future optimizer component.

10.2 Future Research

Many challenging topics for future research remain. Some of them have already been identified in previous chapters.

Advanced Optimizer Currently it is only possible to manually select different operator implementations by using the appropriate planner. This should be enhanced: peers should be able to decide at query processing time which operators are best suited and use them based on local knowledge ("adaptive query processing"). The execution engine already supports this, as it can handle plans which still contain logical operators. Further techniques for local optimizations of MQP-like plans were presented in [PM02a, PM02b] and discussed in sec. 6.6. Some hints are already provided by the evaluation of the operators, and a cost model based on P-Grid's costs for routing of messages could also be integrated.

Additional Operators A proposal for processing $Aggregation/Grouping$ was made in sec. 7.6.1. The presented approaches could be implemented and evaluated. These operations are important for many applications, and processing them in a distributed setting poses some interesting challenges.

In sec. 7.6.2, additional operators to complement the implemented ones were presented[1]. They are relatively easy to implement, and as they can make use of DHT features like range queries and direct hash key lookups, they can speed up many interesting VQL queries.

Improve Centralized Operators `ParallelJoin` and `ParallelRanking` rely on central peers for synchronization. This does not scale very well in a P2P environment, so alternative solutions should be implemented and evaluated.

[1] for example, substring search and optimized equijoins

Further Tests Some areas for future tests are:

- evaluation of parallel join branch execution
- testing with bigger nets
- re-testing of schema similarity queries
- load tests

Bibliography

[Abe01] Karl Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *CoopIS*, 2001. 2.1.3, 2.2, 2.2

[ACMD+03] Karl Aberer, Philippe Cudré-Mauroux, Anwitaman Datta, Zoran Despotovic, Manfred Hauswirth, Magdalena Punceva, and Roman Schmidt. P-Grid: A Self-Organizing Structured P2P System. *SIGMOD Record*, 32(3):29–33, 2003. 2.2

[ADHS05a] Karl Aberer, Anwitaman Datta, Manfred Hauswirth, and Roman Schmidt. Das P-Grid-Overlay-Netzwerk: Von einem einfachen Prinzip zu einem komplexen System. *Datenbank-Spektrum*, 5(13):14–23, 2005. 2.2

[ADHS05b] Karl Aberer, Anwitaman Datta, Manfred Hauswirth, and Roman Schmidt. Indexing Data-Oriented Overlay Networks. In *VLDB*, pages 685–696, 2005. 2.2

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD Conference*, pages 261–272, 2000. 3.1

[Bit07] BitTorrent.org. BitTorrent Protocol Specification. `http://www.bittorrent.org/protocol.html`, January 2007. 1.1, 2.1

[CCR+03] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *SIGCOMM Comp. Comm. Rev.*, 33(3), 2003. 3.4, 9.1

[CF04] Min Cai and Martin R. Frank. RDFPeers: A Scalable Distributed RDF Repository Based on a Structured Peer-to-Peer Network. In *WWW*, pages 650–657, 2004. 3.3

[CFCS04] Min Cai, Martin R. Frank, Jinbo Chen, and Pedro A. Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. *J. Grid Comput.*, 2(1):3–14, 2004. 3.3

[Con07] The P-Grid Consortium. P-Grid Website. `http://www.p-grid.org`, January 2007. 2.2

[DG92] David J. DeWitt and Jim Gray. Parallel Database Systems: The Future

of High Performance Database Systems. *Commun. ACM*, 35(6):85–98, 1992. 3

[DHJ+05] Anwitaman Datta, Manfred Hauswirth, Renault John, Roman Schmidt, and Karl Aberer. Range Queries in Trie-Structured Overlays. In *P2P*, pages 57–66, 2005. 2.1, 2.2

[GIJ+01] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate String Joins in a Database (Almost) for Free. In *VLDB*, pages 491–500, 2001. 4.1.2

[HCH+05] Ryan Huebsch, Brent N. Chun, Joseph M. Hellerstein, Boon Thau Loo, Petros Maniatis, Timothy Roscoe, Scott Shenker, Ion Stoica, and Aydan R. Yumerefendi. The Architecture of PIER: An Internet-Scale Query Processor. In *CIDR*, pages 28–43, 2005. 3.1

[HHL+03] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER. In *VLDB*, pages 321–332, 2003. 3.1

[HHW97] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *SIGMOD Conference*, pages 171–182, 1997. 6.4.1

[Kos00] Donald Kossmann. The State of the Art in Distributed Query Processing. *ACM Comput. Surv.*, 32(4):422–469, 2000. 4.2.2, 6.1

[KSHS06a] Marcel Karnstedt, Kai-Uwe Sattler, Manfred Hauswirth, and Roman Schmidt. Cost-Aware Processing of Similarity Queries in Structured Overlays. In *P2P*, pages 81–89, 2006. 4.2.1

[KSHS06b] Marcel Karnstedt, Kai-Uwe Sattler, Manfred Hauswirth, and Roman Schmidt. Similarity Queries on Structured Data in Structured Overlays. In *ICDE Workshops*, pages 32–37, 2006. 4.1

[KSR+07] Marcel Karnstedt, Kai-Uwe Sattler, Martin Richtarsky, Jessica Müller, Manfred Hauswirth, Roman Schmidt, and Renault John. UniStore: Querying a DHT-based Universal Storage. In *ICDE 2007 Demonstrations Program*, 2007. To appear. 1.1

[LHH+04] Boon Thau Loo, Joseph M. Hellerstein, Ryan Huebsch, Scott Shenker, and Ion Stoica. Enhancing P2P File-Sharing with an Internet-Scale Query Processor. In *VLDB*, pages 432–443, 2004. 3.1

[MSR02] Libby Miller, Andy Seaborne, and Alberto Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. In *International Semantic Web Conference*, pages 423–435, 2002. 3.3

[NWQ+02] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch.

EDUTELLA: A P2P Networking Infrastructure Based on RDF. In *WWW*, pages 604–615, 2002. 2.1.3

[PM02a] Vassilis Papadimos and David Maier. Distributed Queries Without Distributed State. In *WebDB*, pages 95–100, 2002. 6.6, 10.2

[PM02b] Vassilis Papadimos and David Maier. Mutant Query Plans. *Information & Software Technology*, 44(4):197–206, 2002. 3.4, 6.1, 6.2, 6.6, 10.2

[PS06] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Working Draft. `http://www.w3.org/TR/2006/WD-rdf-sparql-query-20061004/`, October 2006. 4.2.1

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *SIGCOMM*, pages 161–172, 2001. 2.1.3, 3.1, 3.4

[RGRK04] Sean C. Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *USENIX Annual Technical Conference*, pages 127–140, 2004. 3.1

[RHS03] Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Range Queries over DHTs. Technical Report IRB-TR-03-009, Intel Research, 2003. 3.1

[RM06] John Risson and Tim Moors. Survey of Research Towards Robust Peer-to-Peer Networks: Search Methods. *Computer Networks*, 50(17):485–521, 2006. 3

[Rös05] Philipp Rösch. Ein Anfrageprozessor für CAN-basierte P2P-Systeme. Diploma thesis, 2005. 3.4

[Sat06] Kai-Uwe Sattler. Verteiltes Datenmanagement. Lecture, TU Ilmenau, 2006. 5.1

[Sch06] Stefan Schwalm. Anfragesystem für vertikal organisierte Universalrelationen in P2P-Systemen. Diploma thesis, 2006. 4.2.1, 4.2.2, 4.2.3, 5.2, 6.3

[SHS05] G. Saake, A. Heuer, and K. Sattler. *Datenbanken — Implementierungstechniken, 2. Auflage*. MITP-Verlag, Bonn, Germany, 2005. 2.1, 2.1.3

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM*, pages 149–160, 2001. 2.1.3, 3.2

[Sop07] SopCast.com. SopCast Introduction. `http://www.sopcast.org/info/sop.jsp`, January 2007. 1.1, 2.1

[TP03] Peter Triantafillou and Theoni Pitoura. Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks. In *DBISP2P*, pages 169–183, 2003. 3.2

[Wie06] Mario Wiegandt. Ähnlichkeitsanfragen in P-Grid-basierten P2P-Systemen. Diploma thesis, 2006. 3.5, 4.1, 4.1.2

[YM98] Clement T. Yu and Weiyi Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. 6.3.1, 6.4

Thesen

1. Im Geschäfts- wie auch im Privatbereich werden immer größere Datenmengen verarbeitet. Aus diesem Grund nimmt die Bedeutung von leistungsfähigen, verteilten Datenbanken zu.
2. Oft wird nicht die komplette Funktionalität klassischer Datenbanksysteme, wie z.B. Transaktionen, benötigt. Der Administrationsaufwand und die Kosten solcher Systeme sind oft zu hoch.
3. Distributed Hash Tables (DHTs) sind massiv skalierbare auf P2P-Technologie basierende Systeme, die zur Verwaltung großer Datenmengen unter Verwendung einfacher Zugriffsfunktionen genutzt werden können.
4. DHTs können um komplexere Anfragebearbeitungsmechanismen erweitert werden ("P2P-Datenbank"), ohne die Skalierbarkeit einzubüßen. Dadurch werden sie für sogenannte Public Data Management-Anwendungen interessant.
5. P-Grid ist eine DHT, die Präfixanfragen und Replikation bietet und eine geeignete Grundlage für eine P2P-Datenbank bildet.
6. Mutant Query Plans (MQPs) sind geeignet für die Implementierung komplexer Anfragebearbeitungsmechanismen in P2P-Systemen.
7. M^2QPs erweitern das MQP-Konzept und ermöglichen eine flexiblere und beschleunigte Bearbeitung von Anfragen in P2P-Systemen.
8. Die Praxistauglichkeit des realisierten Query Processors konnte durch Tests auf PlanetLab mit bis zu 120 Peers unter Beweis gestellt werden.
9. q-gram-Indexe stellen eine geeignete Möglichkeit dar, um Ähnlichkeitsanfragen auf Strings zu beschleunigen.
10. Die Anfragebearbeitung kann erheblich optimiert werden durch die Implementierung eines logischen Operators durch verschiedene physische, die verschiedene Indexe verwenden.

Ilmenau, 13. März 2007

Martin Richtarsky

Affirmation

Affirmation

Hereby I declare that I have written this thesis by myself without any assistance from third parties and that I have exclusively used the indicated literature and resources.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Literatur und Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, wurden als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ilmenau, 13. März 2007

Martin Richtarsky